In this design, we use both the approaches, and record any differences due to these two approaches. We give the user a chance to manually edit the registry/file changes.

Data type definitions

Interface definitions

All of the interfaces used to communicate with the FSRFD are described in the FSRFD LLD document. This LLD document will cover only interfaces with the other modules.

Interfaces with the Builder UI

The public methods of the InstallMon class described below are interfaces to the Builder UI. Specifically these are:

```
void InstallMon::startCapture(PUNICODE_STRING setup_exe,
   PUNICODE_STRING dest, bool upg, FileTable_t *fTbl);
```

This function is called when the user chooses to start monitoring an install. This function is called after the user has entered all the necessary data such as the setup.exe path etc.

`void InstallMon::stopCapture()`

This is called in rare cases when the user wants to terminate a running install in abnormal cases.

`bool InstallMon::checkSetupStatus()`

This is used by the UI to check the status of the install: whether the file list is ready to be processed and the registry list is ready to be processed etc.

`void InstallMon::getRegistryList()`

This function is used to finally get the list of registry changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

`void InstallMon::getFilesList()`

This function is used to finally get the list of file system changes that occurred after the install has taken place. This function allows the user to edit the list to manually add/delete/modify entries to be able to override.

`void InstallMon::machineToBeRebooted()`

When the app install program is asking the user to reboot the machine (in the middle of the install) to continue installation, the user has to inform the Builder UI that a reboot is imminent. The InstallMon does the necessary bookkeeping to make sure that monitoring continues after the reboot.

```
void InstallMon::startCaptureAfterReboot(setup_exe, dest,
upg)
```

When the installation continues after the reboot, the Builder UI is automatically invoked, and it calls this function to inform the InstallMon to continue monitoring.

Access DB interface

We will be using an Access DB (or alternately the MSI databases as suggested by Bhaven) to store intermediate results of the Installmon process. There will be 2 tables used: Registry and Files.

Registry table

```
Fullpath :string;          // this is the full path of the key
ValueName :string;         // this is the value name: for
             // (default) use NULL
UpdateType: char;          // Add/update or delete, also
             // 'R' for removed
// combined (Fullpath, ValueName) should be unique i.e.
// primary key
```

Files table

```
Fullpath : string;
UpdateType : char;         // add, update
Kind: char;                // 'C'opied, 'S'poofed, 'E'stream
FileId: Number;
// Combined (Fullpath, UpdateType, FileId) should be unique
```

RegistryDiff table

```
Fullpath :string;
ValueName:string;
ValueType: char;
Value: something that can store all REG_* types
status: char;   // 'O'riginal, 'C'(add/change), 'D'
// combined (Fullpath, ValueName) should be unique
```

FilesDiff table

```
Fullpath: string;
Type: char;     // file 'F' or dir 'D' etc
Status: char;     // 'O'riginal, ...'C', 'A', 'D' etc
// (Fullpath
```

Component Design

```
struct FileTable_t {
   PUNICODE_STRING name;
   Char type; // spoofed, copied or eFS
   Int  fileId;
};

class InstallMon {
```

```
PUNICODE_STRING setup, destDrive;
bool interruptThread;
EventObject signalMonitor, signalSetup;
bool rebootReq = false;
bool afterReboot = false;
bool completed;
int exitCode;
bool upgrade;
someType envVars;
// TODO: combine the above 2 into an enum
int upgradeFileIdBegin = -1;
int currFileId;
static threadSetup(InstallMon iMon) {
  Remove all the environment variables except
  the ones in iMon->envVars;
  Start the Setup program;
  And wait for it to finish;
  iMon->exitCode = exit code from the process;
  when finished signal the signalSetup event;
}
static threadMonitor(InstallMon *iMon) {
     // this is the func that runs as a separate
     // thread, until we are interrupted or we
     // notice that the setup process has exited.
  get the current
  process id, system drive (using
  GetSystemWindowsDirectory), destDrive and send
  the MON_ACTIVATE message>
  Start threadSetup thread with setup_exe;
  processEnded = false;
  while (!interruptThread) {
       if (rebootReq) {
         only poll for Installmonevent;
         if (signal not set) {
           break from the while loop;
         }
       }
       else {
         WaitForMultipleObjects -> signalMonitor,
         Installmonevent and signalSetup;
       }
       switch (signal) {
       case Installmon event:
         get the MonitorEntry_t record;
         switch (regOrFile) {
         case 'R':
           switch (updateType) {
              case 'A':
              case 'U':
                 add or update (KeyName, ValueName,
                     'N') to
                   registry table;
                 break;
              case 'D':
                 add (KeyName, ValueName, 'D') to
                   registry table;
           }
           break;
        case 'F':
           switch (updateType) {
              case 'A':
                 // file creation
                 add (fullpath, 'A', '?',
                   iMon->currFileId++) to files
                   table;
                 break;
              case 'U':
                 add (fullpath, 'U') to files
                   table;
                 break;
           }
           break;
        case 'E':
           no more data to add;
           if (!processEnded) {
              // something wrong!!!!
           }
           break out of the while loop
           break;
     }
     break;
  case setupevent:
     processEnded = true;
     Send MON_DEACTIVATE to the FSRFD;
     break;
  case signalMonitor event:
     if (rebootReq) {
        kill the threadSetup thread
        clear the signalMonitor event;
     }
     else {
        // TBD?
     }
     break;
```

```
            } // switch
        } // while
        if (processEnded) {
            // normal setup process completion
            // registry and files tables should have
            // all the captured data from the FSRFD
            iMon->diffCapture();
            iMon->completed = true;
        }
    } // threadMonitor
    void commonCapture(setup_exe, dest) {
        setup = setup_exe;
        destDrive = dest;
        Start the threadFunc thread, and pass 'this' to
        it;
    }
    void diffCapture() {
      // Bhaven suggested that we could instead do an export
      // from regedit and do a text diff for registry diffs.
      // similarly: It seems the regular Unix diff tool also
      // compares directories. I don't know if it is
      // recursive. If it is, we might be able to use it.
      // Further investigation is recommended for doing
      // files diff.
      Query the OTI\BUILDERSTART key and get the value
      in builderStartTime and delete the
      OTI\BUILDERSTART key;
      // process the registry
      // step 1: query RegistryDiff table
      for each row in RegistryDiff table {
        query the corresponding key+valueName in the
        Windows registry
        if (exists) {
          if (no change to value) {
            update the row status to 'U' for unchanged
          }
          else {
            update the row status to 'C' (changed)
          }
        }
        else {
          update the row status to 'D' (deleted)
        }
      }
      // step 2: enumerate the whole Windows registry
      for each enumerated registry key+value {
        query the RegistryDiff table;
```

```
   if (a row exists) {
      if (status is 'U') {
         delete the row
      }
      else {
         status should be 'C' and values should be
         different bet table and actual registry
         or else display fatal error(?)
      }
   }
   else {
      // no previous value
      add a row to RegistryDiff with (fullpath,
         valueName, ValueType, Value, 'A') to mark
      it as an added registry key
   }
}
// now repopulate the FilesDiff table
// step 1: query FilesDiff table
for each row in FilesDiff table {
   query the corresponding Fullpath in the actual
   file system;
   if (exists) {
      if (no change (i.e. timestamp before
         builderStartTime)) {
         update the row status to 'U' for unchanged
      }
      else {
         update the row status to 'C' (changed)
      }
   }
   else {
      update the row status to 'D' (deleted)
   }
}
// step 2: traverse the whole file system
for each file/dir in {systemDrive, destDrive} {
   query the FilesDiff table;
   if (a row exists) {
      if (status is 'U') {
         delete the row
      }
      else {
         status should be 'C' and the file/dir
         timestamp should be after builderStartTime
         or else display fatal error(?)
      }
```

```
            }
            else {
               // no previous value
               add a row to FilesDiff with (fullpath,
                  'F' or 'D', 'A') to mark it as an added
               file/dir;
            }
         }
      }
      void recursiveFileGetter(curDir) {
         add a row to FilesDiff as (curDir, 'D', 'O');
         for (each element of curDir) {
            if (element is a dir) {
               recursiveFileGetter(element);
            }
            else { // has to be a file(?)
               add a row to FilesDiff as (element, 'F',
                     'O');
            }
         }
      }
   public:
      InstallMon(?);
      void setEnvVars() {
         allow the user to set environment variables
         in an interactive way;
         Get the values in envVars;
      }
      void startCapture(PUNICODE_STRING setup_exe,
            PUNICODE_STRING dest, bool upg, FileTable_t *fTbl) {
         clear the Access DB registry, files,
            registryDiff and FilesDiff tables;
         if (upg) {
            populate the files table with data from the
               fTbl array;
            upgradeFileIdBegin = last file id + 1;
            currFileId = upgradeFileIdBegin;
         }
         else {
            currFileId = 0 or 1 (depends on where to start);
         }
         Query the whole registry and
         for (each key and valueName pair) {
            add (key, valueName, valueType, value, 'O') to
               registryDiff;
         }
         Get the current time and store it in some format
```

```
    in a registry key OTI\BUILDERSTART;
    for (curDrive in {systemDrive, dest}) do
    {
      Root = root of curDrive (e.g. "C:\\");
      recursiveFileGetter(Root);
    }
    interruptThread = false;
    afterReboot = false;
    commonCapture(setup_exe, dest, upg);
}
void stopCapture() {
      // abnormal capture termination
      // TBD
      // also there might be normal cases where this
      // func is called when the setup process exit
      // is not detected for some reason (or doesn't
      // happen)! May be we don't have to worry about
      // these things.
}
bool checkSetupStatus() {
    if (!completed) {
      display error and return false;
    }
    if (exitCode is not okay) {
      display error and return false;
    }
}
void getRegistryList() {
    checkSetupStatus() and conditionally return;
    Using appropriate SQL, show (fullpath, ValueName)
      tuples that exist in Registry but not in
       RegistryDiff;
    Tell the user that these were captured by FSRFD
    (installmon) driver but not by the diff process;
    If user chooses to remove any tuples from the
    shown list, mark these with status as 'R' in the
    Registry table;
    Using appropriate SQL, show (fullpath, ValueName)
      tuples that exist in RegistryDiff but not in
       Registry;
    Tell the user that these were captured by diff
    but not by the FSRFD (installmon) driver;
    If user chooses to add any tuples from the
    shown list, add these tuples to the Registry
    table with status copied from RegistryDiff;
    Remove all the rows from the Registry table
    where status is 'R';
```

```
      Now we have all the correct entries in Registry;
      Read each row of Registry and into an array to be
      passed to the caller (most probably the caller of this
      func would have passed an array in which we should
      pass these rows);
   }
   void getFilesList() {
      checkSetupStatus() and conditionally return;
      Using appropriate SQL, show (fullpath)
        tuples that exist in Files but not in
        FilesDiff;
      Tell the user that these were captured by FSRFD
      (installmon) driver but not by the diff process;
      If user chooses to remove any tuples from the
      shown list, mark these with status as 'R' in the
      Files table;
      Using appropriate SQL, show (fullpath)
        tuples that exist in FilesDiff but not in
        Files;
      Tell the user that these were captured by diff
      but not by the FSRFD (installmon) driver;
      If user chooses to add any tuples from the
      shown list, add these tuples to the Files
      table with status copied from FilesDiff;
      Remove all the rows from the Files table
      where status is 'R';
      Now we have all the correct entries in Files;

      Using appropriate SQL, select files (and not
      dirs) where the file has only 'U' and not 'A':
      this means the setup modified an existing file
      and we cannot handle this; so if this happens
      show a fatal error;

      Read each row of Files and classify it into 'C',
      'S' or 'E' based on the following logic:
      If the file belongs to the app install directory
      (or app drive?) it is an 'E';
      If the file is smaller than a threshold then
      it is 'C' or else it is 'S';

      if (upg) {
         we need to create new file-ids for directories
         that contain new files or directories
         int prevStart = upgradeFileIdBegin;
         int prevEnd = currFileId;
         while ((prevEnd - 1) > prevStart) {
```

```
        using appropriate SQL, select rows from the files
        table where fileId >= prevStart and kind is not
           'C';
        for each such row {
           fullpath = fullpath in the row; (e.g. "C:\A\B")
           dir = parent directory of fullpath (e.g.
                 "C:\A")
           select in files a row where fullpath = dir and
           fileId >= upgradeFileIdBegin;
           if (doesn't exist) {
              add in Files a row with (dir, 'X', '?',
                currFileId++);
              // Here 'X' stands for "new file id for a
              // directory created because one of the
              // children has a new file id
           }
        }
        prevStart = prevEnd;
        prevEnd = currFileId;
    }
}
```

Also for each file, change the absolute path to an envVar relative or registry-value relative one: this is a non-trivial task. The way to do this is proposed below. However we need to refine this algorithm based on actual Builder runs on real apps:

For a basic Win2K (or WinNT as the case may be) system create a list of registry keys (and env vars) whose values are normally used as destination paths for copying various kinds of files. To this list also add this app's dest dir as one of the values. Also add all the registry keys/values added as part of this app's install. Create an access table that looks like:

```
Key: String;     // registry or env or other name
Type: char;     // 'R'egistry, 'E'nv, 'D'est dir etc.
Source: char;   // 'S'ystem, 'A'pp, 'U'nknown?
Value: string;
```

Now all of the 'E' files should be relative to dest dir (i.e. 'D' type above). Also any sub-directory strings should either be hard-coded or based on some registry key values where the registry key is of source 'A' (app) above.

```
   All the 'C' and 'S' files should be relative to one of
   the registry keys of source 'S' above and preferably
   type 'R' (since environment variables are not used un-
   der Windows?).

   If there is a sacred file set (i.e. files that are
   so "sacred" for eStream apps, that they were installed
   on the client when the eStream client was installed),
   we need to remove those files from this list;

   Set these values in the table and read the whole
   table in an array (most probably the caller of
   this func would have passed an array in which we
   should pass these rows);
}
void machineToBeRebooted() {
      //The setup is going to reboot the machine.
      //Note that there is a race condition possible here.
      // Suppose the setup.exe has displayed a dialog
      // asking the user to confirm a reboot. At this time
      // the setup.exe has still not written to "Runonce".
      // Only after the user has confirmed (by pressing
      // Okay) will the program write to "runonce" and
      // auto reboot. In that case, this function will be
      // called at the wrong time. We may need to modify
      // the FSRFD to detect changes to "Runonce".
      rebootReq = true;
      signal the signalMonitor event;
      wait for the threadMonitor thread to end;
      Query either our Access database or the actual
      registry to see if Runonce key is set/updated.
      if (not) {
         we cannot handle this reboot situation;
         give fatal error;
         // may be we can look at the
         // Start\Programs\Starup folder and do the
         // same thing we did with Runonce key
      }
      OldRunonce = Old value of Runonce key;
      Set the new value of Runonce key as our Builder
      name (or whichever EXE has the installmon code)
      followed by OldRunonce as the argument to that
      EXE and the destDrive as the next argument and
      upgrade as the next arg;
      Prompt the user to go ahead and say Okay to
      Setup's dialog warning that it is going to
      reboot;
   }
   void startCaptureAfterReboot(setup_exe, dest, upg) {
      afterReboot = true;
      commonCapture(setup_exe, dest, upg);
   }
};
```

Interesting issues to deal with:

Testing design

Unit testing plans

The unit testing of installmon will be done in conjunction with the FSRFD unit testing. This has been described in the FSRFD-LLD document. In addition, we will be using the sysdiff tool to validate the results of the Installmon.

Stress testing plans

All of the 14 or so applications (that OTI is planning to convert to eStream) installs will be tested under the installmon. Any issues found will be used to fix and improve the installmon functionality.

Coverage testing plans

The testing of the Builder/installmon on the 14 or so app installs should give us enough coverage. Considering that Builder/installmon will be used in-house for some time makes some of the testing issues less significant.

Cross-component testing plans

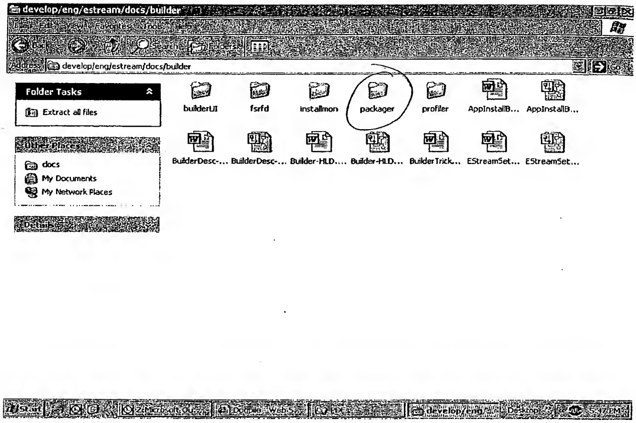
Will be tested as a component of the whole builder.

Upgrading/Supportability/Deployment design

Deployment: This will be used in-house, so no deployment considerations.

Open Issues

- On one of the newgroups someone mentioned a key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs] that we can look at for the shared DLLs that cannot be installed. Need to investigate.
- 2-phase install: some installs need a reboot after which they continue. The key to look at is Runonce (under the same path as SharedDLLs I think).



[/STANDARD.01]

[redacted]

eStream Builder Package Manager Low Level Design

Sanjay Pujare and David Lin

Version 0.1

[redacted]

Functionality

The eStream Application Builder Package Manager is responsible for packaging data gathered from the Installation Monitor, the Profile Manager, and the Upgrade Monitor into a set of data called the eStream Set. For the detail format of the eStream Set, see the separate document on eStream Set.

The Package Manager must perform the following task:

- Create the appInstallBlock containing C-File and Registry data from the Install Monitor; Prefetch data from the Profile Manager; and Updated C-File and Updated Registry data from the Upgrade Monitor
- Create a custom installation DLL needed by a specific applications and add to the appInstallBlock
- Create directory files associated with each directory of the application director and add metadata to the directory
- Create directory files associated with each Windows directory containing both the Spoofed files and Z-files
- Create Concatenated Application File (CAF) which is just a juxtaposition of the application files, eStream directory files, and AppInstallBlock
- Create Size Offset File Table (SOFT) which is a mapping of fileNumber to offset of the start of the CAF file
- Create Root Version Table (RVT) which is a mapping from the version of root to the file number of the root directory file
- Archive the CAF, SOFT, and RVT into a single structure called eStream Set suitable for uploading to the eStream Servers.

Data type definitions

The Package Manager doesn't have any internal data types. It must accept and understand data structures received from the Install Monitor and the Profile Manager. See Install Monitor and Profile Manager components for the description of the data structures.

The Install Monitor is responsible for generating the following list of information: list of copied-files, list of spoof-files, list of files with file numbers, list of add registry entries, and list of delete registry entries. The list of copied-files contains the files copied into non-application specific directories. The list of spoof-files consists of the files too large to be downloaded to the client in the AppInstallBlock. Those files are copied into some special directory on the Z drive for streaming. The list of files with file numbers consists of the files copied into the standard "Program Files" directory and the files that will be spoofed. The registry information is a list of registry key added or removed during the installation of the application.

```
Struct FileIndexTable {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
      ULONG FileNumber;
   } Entries[NumEntries];
};
Struct FileCopied {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING FilePathName;
   } Entries[NumEntries];
}
Struct FileSpoofed {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING OldFilePathName;
      PUNICODE_STRING NewFilePathName;
   } Entries[NumEntries];
};
Struct RegistryInfo {
   UINT NumEntries;
   Struct Entry {
      PUNICODE_STRING KeyName;
      PUNICODE_STRING ValueName;
      PVALUE_DATA ValueData;
   } Entries[NumEntries];
};
Struct IniInfo {
   UINT NumFiles;
   Struct FileEntry {
      PUNICODE_STRING FilePathName;
      UINT NumSections;
      Struct SectionEntry {
         PUNICODE_STRING SectionName;
         UINT NumValues;
         Struct Entry {
            PUNICODE_STRING ValueName;
            PVALUE_DATA ValueData;
         } Entries[NumValues];
      } Entries[NumSections];
```

```
    } Entries[NumFiles];
};
```

The Profile Manager generates AccessCounts and the PrefetchBlocks data with the structures shown below.

```
Struct AccessCounts {
    UINT NumEntries;
    Struct Entry {
        PUNICODE_STRING FilePathName;
        ULONG Frequency;
    } Entries[NumEntries];
};
Struct PrefetchBlocks {
    UINT NumEntries;
    Struct Entry {
        PUNICODE_STRING FilePathName;
        ULONG BlockNumber;
    } Entries[NumEntries];
};
```

The eStream Set has the following data structure (described in more detail in the separate eStream Set document):

```
Struct eStreamSet {
    Struct eStreamSetHeader header;
    Struct eStreamSetRVT rvt;
    Struct eStreamSetSOFT soft;
    Struct eStreamSetCAF caf;
};
```

Interface definitions

Function 1 : CreateEStreamSet

```
// Create the initial eStream Set from the data
// retrieved from the Install Monitor and the
// Profile Manager.
// This function is called only by the Builder
// UI after data is obtained from Install
// Monitor and Profile Manager.
int CreateEStreamSet(
    IN PFILE_INDEX_TABLE FIT,
    IN PFILE_SPOOFED SpoofFiles,
    IN PFILE_COPIED CopiedFiles,
    IN PREGISTRY_INFO AddRegistry,
    IN PREGISTRY_INFO RemoveRegistry,
    IN PINI_INFO IniInfo,
    IN PACCESS_COUNTS AccessCounts,
    IN PPREFETCH_BLOCKS PrefetchBlocks,
    IN PVOID DllCode,
    IN PUNICODE_STRING Comment,
    OUT PESTREAM_SET EstreamSet)

Input:
    FIT: File Index Tree contains the file
       number of the directories, spoofed
       files, and standard files

    CopiedFiles: pointer to a list of files
       To be copied to AppInstallBlock

    SpoofFiles: pointer to a list of files
       To be spoofed on the client

    AddRegistry: pointer to a list of registry
       Data to add
    RemoveRegistry: pointer to a list of
       Registry data to remove
    IniInfo: pointer to a list of ini changes

    AccessCounts: pointer to the list of
       Files with the access frequency

    PrefetchBlocks: pointer to the prefetch data
       To be inserted into the appInstallBlock
       Of the eStream Set

    DllCode: pointer to DLL Code

    Comment: pointer to comment string

Output:
    EstreamSet: pointer to the eStream Set

Return Value:
    Success or failure of the packaging process

Comments:
    The eStream Set will be large for most
    application.  Intermediate data will be
    stored on the local hard-drive.

Errors:
    OutOfStorage: failure to find enough storage
       For this eStream Set
```

```
FileNotFound: failure to find the files
   Specified by either ListCFiles or
   ListZFiles
```

Function 2 : UpgradeEStreamSet

```
// Upgrade the eStream Set to the latest
// version. This function is only called by
// the Upgrade Manager within the same process.

int UpgradeEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE UpgFIT,
   IN PFILE_SPOOFED UpgSpoofFiles,
   IN PFILE_COPIED UpgCopiedFiles,
   IN PREGISTRY_INFO UpgAddRegistry,
   IN PREGISTRY_INFO UpgRemoveRegistry,
   IN PACCESS_COUNTS UpgAccessCounts,
   IN PPREFETCH_BLOCKS UpgPrefetchBlocks,
   IN PVOID UpgDllCode,
   IN PUNICODE_STRING UpgComment)

Input:
   UpgFIT: File Index Tree contains the file
      number of the directories, spoofed
      files, and standard files

   UpgCopiedFiles: pointer to a list of files
      To be copied to AppInstallBlock

   UpgSpoofFiles: pointer to a list of files
      To be spoofed on the client

   UpgAddRegistry: pointer to a list of
      Registry data to add

   UpgRemoveRegistry: pointer to a list of
      Registry data to remove

   UpgAccessCounts: pointer to the list of
      Files with the access frequency

   UpgPrefetchBlocks: pointer to the prefetch
      Data to be inserted into the
      AppInstallBlock Of the eStream Set

   UpgDllCode: pointer to DLL Code
   UpgComment: pointer to comment string

Output:
   EstreamSet: pointer to the eStream Set

Return Value:
   Success or failure of the packaging process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
   OutOfStorage: failure to find enough storage
      For this eStream Set

   FileNotFound: failure to find the files
      Specified by either ListCFiles or
      ListZFiles
```

Function 3 : ModifyEStreamSet

```
// Insert new data into different sections
// of estream set.  This function is overloaded
// to handle different section data

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE FIT,
   IN PPREFETCH_BLOCKS PrefetchBlocks)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PREGISTRY_INFO AddRegistry,
   IN PREGISTRY_INFO RemoveRegistry)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PFILE_INDEX_TABLE FIT,
   IN PFILE_SPOOFED SpoofFiles,
   IN PFILE_COPIED CopiedFiles)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PVOID DllCode)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PUNICODE_STRING Comment)

int ModifyEStreamSet(
   INOUT PESTREAM_SET EstreamSet,
   IN PUNICODE_STRING LicenseText)

Input:
   FIT: File Index Tree contains the file
      number of the directories, spoofed
      files, and standard files

   CopiedFiles: pointer to a list of files
      To be copied to AppInstallBlock

   SpoofFiles: pointer to a list of files
      To be spoofed on the client

   AddRegistry: pointer to a list of registry
      Data to add
   RemoveRegistry: pointer to a list of
      Registry data to remove
   IniInfo: pointer to a list of ini changes

   AccessCounts: pointer to the list of
      Files with the access frequency

   PrefetchBlocks: pointer to the prefetch data
      To be inserted into the appInstallBlock
      Of the eStream Set

   DllCode: pointer to DLL Code

   Comment: pointer to comment string

Output:
   EstreamSet: pointer to the new eStream Set

Return Value:
   Success or failure of the insertion process

Comments:
   The eStream Set will be large for most
   application.  Intermediate data will be
   stored on the local hard-drive.

Errors:
```

```
OutOfStorage: failure to find enough storage
   For this eStream Set

FileNotFound: failure to find the files
   Associated with the prefetch blocks
```

Component design

The pseudo-code for the function *CreateEStreamSet* is described below:

```
{
    Create AppInstallBlock (AIB) from the following input files:
            o SpoofFiles
            o CopiedFiles
            o AddRegistry
            o RemoveRegistry
            o Prefetch
            o Comment
            o DLLcode
    Assign AppInstallBlock with a unique fileNumber given by the IM;
    Record Root fileNumber in the first entry of Root fileNumber Table (RFT);
    Move AppInstallBlock under the Root directory by adding a new entry in the
        Directory structure;
    Create a Concatenation Application File (CAF) header;
    Create a Size Offset File Table (SOFT) header;
    For each (file in FIT) {
        If (file is a directory) {
            Create the directory with new list of fileNumber, filename, and
                Metadata;
        } Else {
            Find the file in the proper location on the HD;
        }
        Append the file or directory to the end of the CAF file;
        Append the fileNumber, offset into CAF, and size of file in SOFT;
    }
    Archive CAF, SOFT, and RFT into a single eStream Set;
    Return eStream Set;
}
```

The pseudo-code for the function *UpgradeEStreamSet* is mentioned below:

```
{
    Extract previous version PrevAppInstallBlock from eStream Set;
    Create new AppInstallBlock with new FileNumber;

    Extract PrevSpoofFiles and PrevCopiedFiles from PrevAppInstallBlock;
    Divide the C-Files into SpoofFiles and CopiedFiles;
    Add PrevSpoofFiles to SpoofFiles;
    Add PrevCopiedFiles to CopiedFiles;

    Extract PrevAddRegistry and PrevRemoveRegistry data from
        PrevAppInstallBlock;
    Add any unique ((UpgAddRegistry plus PrevAddRegistry) minus
        UpgRemoveRegistry) in the new AppInstallBlock AddRegistry section;
    Add any unique ((UpgRemoveRegistry plus PrevRemoveRegistry) minus
        UpgAddRegistry in the new AppInstallBlock;

    Add UpgPrefetch data to new AppInstallBlock;
    Add UpgDllCode data to new AppInstallBlock;
    Add UpgComment data to new AppInstallBlock;

    For each (directory in UpgFIT) {
        If (any child fileNumber has changed) {
            Create new directory with updated fileNumber;
            Append file to end of Concatination Application File (CAF);
            Append Size Offset File Table (SOFT) with new entry;
        }
    }
    Append new AppInstallBlock to the end of CAF file;

    Prepend Root FileNumber Table (RFT) with new Root entry;

    Archive CAF, SOFT, and RFT into a single eStream Set;
    Return eStream Set;
}
```

The pseudo-code for the overloaded function *ModifyEStreamSet* is mentioned below:

```
{
    // not needed unless merging of uploaded profile data is supported
}
```

Testing design

This document must have a discussion of how the component is to be tested.

- **Unit testing plans**

  The plan for unit testing Package Manager includes the development of a driver program. This driver interfaces to the Package Manager and invokes the functions with different parameters. The list of possible cases is described below:

  1. Test all interfaces by driving the input parameters with different type of add and remove registry values.
  2. Test all interfaces by driving the input parameters by varying numbers of spoof and copied files.
  3. Test all interfaces by driving the input parameters with some prefetch information.
  4. Test all interfaces for meaningless input values from the IM and PM.
     - Prefetch block containing file number not assigned by IM.
     - IM assigning non-continguous file numbers.
  5. Test upgrade interface for capability to detect and handle bad eStream Set gracefully.
  6. Test upgrade interface and make sure it can detect overlapping file number assignments.
  7. Test upgrade interface and make sure prefetch blocks are not referencing old file number from previous versions.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  The output data from the Package Manager is called the eStream Set. This eStream Set is the input to a stand-alone test program called the *eStream Extractor*. The Extractor unpacks and 'install' the eStream Set into the local machine without an eStream client file system installed. This test is used to quickly verify that the eStream Set can be run on a pristine machine. Some of the possible variations of the Extractor test includes:

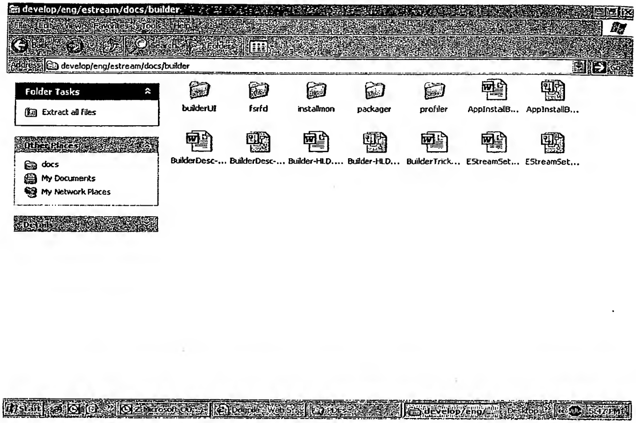
  1. Non-default system variable names. I.e. %SystemRoot% set to "D:\Win" instead of "C:\Winnt".
  2. Non-default eStream FS drive letter. Use Y instead of Z.

Upgrading/Supportability/Deployment design

The Package Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- Which Builder component creates the installation DLL when the application needs the custom installation code? Is a new component needed to create the custom DLL separately and insert into AppInstallBlock in the eStream Set as needed?


develop/eng/estream/docs/builder

Address: develop/eng/estream/docs/builder

builderUI
fsrfd
installmon
packager
profiler
AppInstallB...
AppInstallB...
BuilderDesc-...
BuilderDesc-...
Builder-HLD....
Builder-HLD...
BuilderTrick...
EStreamSet...
EStreamSet...

[/STANDARD.01]

eStream Builder File Access Monitor Low Level Design

Sanjay Pujare and David Lin
Version 0.1
[redacted]

Functionality

The eStream Application Builder File Access Monitor (FAM) is a kernel-mode device driver that behaves as a file filter driver to has the following responsibilities:

- Monitor any running application's request to access a file or directory
- Track application file and directory accesses
- Track file metadata queries
- Start and stop profiling via IOCTL requests from the user-mode program
- Return the file access data to the user-mode program via I/O Request Packet (IRP)
- Return any error conditions to the user-mode program via IRP

The File Access Monitor is based on the 'Filemon' program. The source code for the program is available free for download over the Web at http://www.sysinternals.com/filemon.htm.

Data type definitions

The File Access Monitor (FAM) monitors a sequence of file block accesses by a particular process or one of its child processes. The FAM also tracks any queries on the file metadata. The combination of the file content and metadata is returned to the Profile Manager for further processing. The following is the data structure externally visible to the other subcomponents outside FAM.

```
Struct SequenceData
{
  UINT NumEntries;
  Struct Entry
  {
    PUNICODE_STRING FilePathName;
    BOOL IsAccessingMetadata;
    ULONG Offset;
    ULONG Size;
  } Entries[NumEntries];
};
```

The FileName contains the null terminating string of the file accessed. IsAccessMetadata flag indicates if the access is on the file metadata or the file content. If the operation is on the file content, then the fields 'Offset' and 'Size' indicate the location of the read or write operations. Otherwise, the fields 'Offset' and 'Size' are not used.

Interface definitions

Function 1 : Hooks into user defined IOCTL calls

```
// The following is a Fast I/O Device Control
// call interface.  Each of the user IOCTL
// call to the driver is described here.
// The OICTL input and output parameters are
// stored on the IRP on the InputBuffer and
// OutputBuffer respectively.
// This function must be called only by NT I/O
// Manager.

BOOLEAN FileSysFastIoDeviceControl(
   IN PFILE_OBJECT FileObject,
   IN BOOLEAN Wait,
   IN PVOID InputBuffer,
   IN ULONG InputBufferLength,
   OUT PVOID OutputBuffer,
   IN ULONG OutputBufferLength,
   IN ULONG IoControlCode,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Input:
   IoControlCode==IOCTL_FAM_VERSION
      OutputBuffer: version number of the driver

   IoControlCode==IOCTL_FAM_START
      InputBuffer: process ID to monitor

   IoControlCode==IOCTL_FAM_STOP
      OutputBuffer: stop profiling

   IoControlCode==IOCTL_FAM_GETDATA
      OutputBuffer: get sequence data

   IoControlCode==IOCTL_FAM_GETSTATUS
      OutputBuffer: get status from driver

Output:
```

```
Comments:
   The IOCTL calls from the user program to
   the device driver is either through the
   dispatcher or through the Fast I/O
   interface.

Errors:
```

Function 2: DriverEntry

```
// Called by the NT system to initialize
// driver.  The following entries are hooks
// into the OS and are not called by any of our
// component directly.
NTSTATUS DriverEntry(
   IN PDRIVER_OBJECT DriverObject,
   IN PUNICODE_STRING RegistryPath)

Comments:
   Initialize the driver
```

Function 3: Hooks into Fast I/O functions

```
// NT Fast I/O calls.  These are some of the
// hooks into the OS
NTSTATUS FileSysFastIoRead(
   IN PDRIVER_OBJECT DriverObject,
   IN PLARGE_INTEGER FileOffset,
   IN ULONG Length,
   IN BOOLEAN Wait,
   IN ULONG LockKey,
   OUT PVOID Buffer,
   OUT PIO_STATUS_BLOCK IoStatus,
   IN PDEVICE_OBJECT DeviceObject)

Comments:
   Hooks into Fast I/O Read
```

Function 4: Hooks into dispatcher functions

```
// Besides hooks into Fast I/O calls, we
// must also hook into each of the major
// functions like IRP_MJ_CREATE, IRP_MJ_READ,
// etc...
FileSysHookRoutine(
   PDEVICE_OBJECT HookDevice,
        IN IRP Irp)
```

Component design

I/O Hook location

The trickiest part of the File Access Monitor (FAM) component design is determining the locations in the Operating System to hook the routines. FAM must provide driver entry function for initializing the driver. It must also provide hooks into the OS to monitor read and write file operations. In addition, it needs to monitor access to file metadata. And finally, it has to provide the user-mode program a way to communicate to the FAM through the IOCTL calls.

The FAM must behave like any other Windows kernel-mode drivers by exporting the standard *DriverEntry* function. The *DriverEntry* function has the following purposes:

- Check for OS build version.
- Setup the device name
- Call OS routine to create the device
- Make symbolic link to allow device access from Win32 programs .
- Create dispatch points for all routines that must be handled
- Setup Fast I/O hooks
- Initialize all data structures

In addition to the *DriverEntry*, the FAM must handle five user defined IOCTL calls.

- IOCTL_FAM_VERSION
- IOCTL_FAM_START
- IOCTL_FAM_STOP
- IOCTL_FAM_GETDATA
- IOCTL_FAM_GETSTATUS

In IOCTL_FAM_START, the handler receives the process ID from the user-mode program. It uses this process ID to filter out relevant file and metadata accesses. In IOCTL_FAM_STOP, the handler stops monitoring and recording any file accesses. In IOCTL_FAM_GETDATA, the handler packages the file access sequence in the I/O Request Packet (IRP) to be returned to the user-mode program. Finally, in IOCTL_FAM_GETSTATUS, the handler returns its current status. This status includes: FAM_STATUS_OK, FAM_STATUS_ERROR, and FAM_STATUS_PROFILING.

In addition to the user defined IOCTL hooks, the FAM must add hook into both the dispatch points and the Fast I/O calls to monitor all read and write requests. In addition, the FAM monitors any metadata accesses. The following is a list of Fast I/O calls it must hook:

- FastIoRead
- FastIoWrite

- FastIoMdlReadComplete
- FastIoMdlWriteComplete
- FastIoReadCompressed
- FastIoWriteCompressed
- FastIoQueryBasicInformation
- FastIoQueryStandardInformation

In the routine to handle FastIoRead and FastIoWrite, the driver must determine the process ID making this request. If the process is in the list of monitoring processes, the file name, file offset, and size is recorded and added to the profile sequence list. In the routine to handle FastIoQueryBasicInformation and FastIoQueryStandardInformation, the driver records the file name associated with this metadata query.

In addition to hooks to the Fast I/O calls, the I/O may call the File System services through standard Windows NT dispatch points. The following is a list of dispatch points to be handled by FAM:

- IRP_MJ_CREATE
- IRP_MJ_READ
- IRP_MJ_WRITE
- IRP_MJ_DIRECTORY_CONTROL + IRP_MN_QUERY_DIRECTORY
- IRP_MJ_QUERY_INFORMATION
- IRP_MJ_SET_INFORMATION
- IRP_MJ_QUERY_EA
- IRP_MJ_SET_EA

The routine to handle IRP_MJ_READ, IRP_MJ_WRITE, and IRP_MN_QUERY_DIRECTORY is handled by the same function as the routine for handling FastIoRead and FastIoWrite. The routine to handle IRP_MJ_QUERY_INFORMATION, IRP_MJ_SET_INFORMATION, IRP_MJ_QUERY_EA, and IRP_MJ_SET_EA are handled by the same function as the routine for handling FastIoQueryInformation.

Communication with user-mode component (Profile Manager)
Besides using the IOCTL to send profile data to the Profile Manager, the FAM must also signal the Profile Manager when new data is available for retrieval. The Profile Manager wakes up from the signal by the FAM and retrieves the information on the blocks of files accessed. FAM also signals the profile manager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWaitForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Process Filtering
The FAM must filter the profile information so only relevant data relating to the application under profiled is obtained. This is accomplished by filtering the data according to

the process ID invoking the file access operations. When the FAM is started, the Profile Manager sends its process ID. FAM assumes all child processes of the Profile Manager process ID is to be monitored since the Profile Manager invokes all applications using *CreateProcess()* API. Thus, the new processes all inherit Profile Manager as its parent process ID. The process filtering is accomplished using *PsSetCreateProcessNotify-Routine()* to add a hook to the OS. FAM is notified whenever there is a new process created. The process ID is recorded in a list if its ancestor is the Profile Manager process ID. This list is used to filter the profile data gathered by FAM.

Locks
Since multiple threads may be entering different sections of FAM and accessing different data structures, appropriate locks must be used to prevent multiple threads from reading and writing at the same time. *ExInitalizeResourceLite(), ExAcquireResourceExclusiveLite(),* and *ExReleaseResourceLite()* are used when shared data structure is accessed. These APIs have the requirement that the kernel APCs must be disabled before calling and that IRQL must be lower than DISPATCH_LEVEL. This can be accomplished by using *KeEnterCriticalRegion()* and *KeLeaveCriticalRegion()*. The following is a sample code using these APIs:

```
ERESOURCE gResource; // global variable

KeEnterCriticalRegion()
ExAcquireResourceExclusiveLite(&gResource, TRUE);

<critical section of code>

ExReleaseResourceLite(&gResource);
KeLeaveCriticalRegion();
```

Testing design

- **Unit testing plans**

  The plan for unit testing of the FAM consists of using the Profile Manager (PM) and a File Access Driver (FAD) as the test drivers. The PM tests user-defined IOCTL calls. The FAD creates desired data pattern from the OS's I/O Manager to the FAM. The FAD tests the FAM's ability to monitor file accesses by querying files and directories in a particular order. Together, the PM and FAD test coverage of the FAM is complete. The following is a list of tests:

  1. Test each user-defined IOCTL interface via PM by sending border cases.
  2. Test to make sure FAM captures every file and directory access via standard file I/O requests from a user-mode program called FAD.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

Other Builder components log error messages to a predefined file. The kernel-mode programs do not have the capability to read/write to a file. Since FAM is a kernel-mode program, an alternative method of reporting error messages has to be developed. Current, the FAM has a user-defined IOCTL interface (IOCTL_FAM_GETSTATUS) to retrieve the error messages. FAM keeps a stack of error messages encountered and reports the stack of error messages at the request by an appropriate user-mode program.

Open Issues

- Exactly which Fast I/O calls need to be hooked to get all the read and write operations for file accesses?
- Along the same line, which dispatch points need to handled to get all the read and write operations for file accesses?
- Have we hooked into all possible places where the metadata accesses can occur?
- Does the FAM need to hook into FileLock and FileUnlock operations?

eStream Builder Profile Manager Low Level Design

Sanjay Pujare and David Lin
Version 0.3
[redacted]

Functionality

The eStream Application Builder Profile Manager is responsible for the following:

- Receive request from the UI Component for one or more application executables to profile.
- Accumulate each run of the profile data in a data structure suitable for merging.
- Invoke each application executable for a fixed amount of time, for a fixed number of prefetch blocks, for a simple start-stop of the program, or for multi-level profiling based on scripts or manual usage of an application.
- Communicate with File Access Monitor (FAM) kernel-mode driver using IOCTLs to start and stop profiling.
- Obtain the complete file access sequence data from the FAM.
- Process the file access sequence into two parts: a table of file access frequency and a list of prefetch blocks.
- Send the resulting profile data to Package Manager component for integration into the AppInstallBlock.

This component will probably exist as a class object and will be instantiated by the Builder user interface component. The component will run in the same process as the user interface component. Please see Builder User Interface component document for more information on that component.

Data type definitions

The Profile Manager imports the file access sequence from the FAM. The data structure is described below: (Please see the File Access Monitor for detail information on the meaning of each field in the data structure)

```
Struct SequenceData
{
   UINT NumEntries;
   Struct Entry
   {
      UNICODE_STRING FilePathName;
      BOOL IsAccessingMetaData;
      ULONG Offset;
      ULONG Size;
   } Entries[NumEntries];
};
```

Profile Manager creates two data structure from the data received from the Install Monitor and the FAM. The consumer of the output data structure is the Package Manager. These data structures is described in the following two structures:

```
Struct PrefetchBlockList {
   UINT NumSections;
   Struct PrefetchBlocks {
      UINT BlockType;
      UINT NumEntries;
      Struct Entry {
         UNICODE_STRING FilePathName;
         ULONG BlockNumber;
      } Entries[NumEntries];
   } Entries[NumSections];
};
Struct ProfileApplications {
   UINT NumEntries;
   Struct Entry {
      UNICODE_STRING FilePathName;
      UNICODE_STRING Arguments;
   } Entries[NumEntries];
};
```

The access count is used to order the list of files in a directory according to metadata file access frequency. The prefetch data is encorporated into the AppInstallBlock by the Package Manager to be used by the eStream Client.

Interface definitions

Function 1 : StartProfiling

```
int StartProfiling(
   IN PPROFILE_APPLICATIONS AppList,
   IN UINT Type,
   IN UINT TypeData,
   OUT PPREFETCH_BLOCKS PrefetchBlocks)

Input:
   AppList: a list of file pathnames and
      Arguments to run the application.

   Type: type of profiling to do
      SIMPLE: start and stop application
      TIMEBASED: profile for a fixed time and
```

```
         terminate application
      SIZEBASED: profile for a fixed size of
         Profile data

   TypeData: extra data for profile type
      If ProfileType==SIMPLE, TypeData is
         Ignored
      If ProfileType==TIMEBASED, TypeData is
         Length of time in seconds
      If ProfileType==SIZEBASED, TypeData is
         Size of profile data in bytes
      If ProfileType==COMMANDBASED, TypeData is
         Pointer to a possible list of script
         Files to be invoked

   PrefetchBlocks: List of prefetch blocks
      To add to the AppInstallBlock

Return value:
   Success or failure code of the profiling

Comments:
   The profile manager component actually send
   IOCTLs to the file filter device driver to
   Start and stop the gathering of the profile
   Data and to retrieve the profile data.

Errors:
   FileNotFound: some of the application
      Executables in the list may not exist or
      Not readable
   ProfileTimeout: failed to gather the desired
      Size of the profile data after certain
      Amount of time
   DriverFailure: File Access Monitor return
      Failure code to the Profile Manager which
      Must propagate the error to the user
      Interface
```

Component design

Application Invocation

To start profiling, the component must have a list of application pathname to be invoked. The pathname may be an executable with a list of arguments. Or the pathname may be a Windows short-cut file. If the pathname is an exe file, then the standard Win32 API *CreateProcess()* is used. On the other hand, if the file is a Windows short-cut file, then the component needs to extract relevant information from the short-cut file to make the proper *CreateProcess()* invocation. The following is a pseudo-code for extracting pathname and argument information from the short-cut file using IShellLink interface:

```
GetInfoFromShortCutFile(char *strPath, char *strArg)
{
   IshellLink *psl;
   WIN32_FIND_DATA fd;
   if (SUCCEEDED( CoCreateIstance(CLSID_ShellLink,
                  NULL,
                  CLSCTX_INPROC_SERVER,
                  IID_IShellLink,
                  (LPVOID*) &psl)))
   {
     psl->GetPath(strPath, MAX_LEN, &fd, 0);
     Psl->GetArguments(strArg, MAX_LEN);
     psl->Release();
   }
}
```

Command-based Profiling
One of the options for profiling includes the ability to identify blocks of files accessed when specific application command is invoked. The profiler prompts the user for the desired actions (ie. Open document, save document, etc) and gathers file blocks that are accessed corresponding to those actions. These commands are saved into the AppInstall-Block for eStream client to intelligently pick the proper set of blocks to stream. The following is an enumeration of some divisions of prefetch blocks:

- Start Application
- End Application
- Save Document
- Open Document
- Cut Sections
- Copy Sections
- Paste Sections

In the current design, the profiler divides the code into two prefetch sections. The first section contains the critical blocks necessary for efficient startup of the streamed application for the very first time. The blocks of code in this section include starting of application. The second section is the common blocks necessary for efficient running of the streamed application. This includes common user operations like opening and saving of document.

Communication with kernel-mode driver (FAM)
The Profile Manager communicates with the kernel-mode driver to retrieve the information on the blocks of files accessed. The profile manager waits for a signal from the FAM indicating a new data is available for retrieval. FAM also signals the profile man-

ager when the profiled application terminates. FAM uses *KeSetEvent()* to send a 'data available' event signal to the profile manager. Profile manager calls *KeWait-ForSingleEvent()* or *KeWaitForMultipleEvent()* to wait for a signal from the kernal-mode driver. *KeClearEvent()* is called by the FAM when the signal to profile manager should be deactivated.

Pseudo-code
The pseudo-code for the function *StartProfiling* is described below:

```
{
    Initialize GlobalFileAccessCounts and GlobalPrefetchBlocks;
    Load FAM if not loaded;
    For each (executable in the list of application) {
        Start FAM by sending it process ID of the Profile Manager;
        Create new process for this executable and run it;
        Switch (Type of Profiling) {
        Case SIMPLE:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until application start up;
            Break;
        Case TIMEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } Until fixed time unit;
            Break;
        Case SIZEBASED:
            Loop {
                Wait for an event from FAM;
                Get Status from FAM;
                Get SequenceData from FAM;
            } while (size < fixed amount);
            Break;
        Case COMMANDBASED:
            For each command in the list {
                If (script exist)
                    Run script on the executable program;
                Else
                    Prompt operator for proper action;
                Loop {
                    Wait for an event from FAM;
                    Get Status from FAM;
                    Get SequenceData from FAM;
                } Until script completed;
            }
        }

        Send WM_QUIT message to the application process;
        Loop {
            Wait for an event from FAM;
            Get Status from FAM;
            Get SequenceData from FAM;
        } Until application quit;
        Inform FAM to stop gathering profile data;

        Compute PrefetchBlocks from the SequenceData and append to
            GlobalPrefetchBlocks;
    }
    Unload File Access Monitor (if possible);
    Return GlobalPrefetchBlocks;
}
```

Testing design

- **Unit testing plans**

The plan for unit testing the Profile Manager includes developing a driver to connect to the interface between the Profile Manager and the Builder UI. The driver conducts the following types of tests:

1. Test different type of profiling including simple profiling, time-based profiling, size-based profiling, and script-based profiling.

2. Test different executable programs and make sure the output data "makes sense".
3. Test a list of executables for merging capability of the Profile Manager.
4. Test the interface between Profile Manager (PM) and the File Access Monitor (FAM) using FAM as the test driver. The FAM can check for any valid IOCTL calls from the PM. FAM can also reply to IOCTL calls with different values in the IRP to simulate all possible cases.

- **Stress testing plans**

- **Coverage testing plans**

- **Cross-component testing plans**

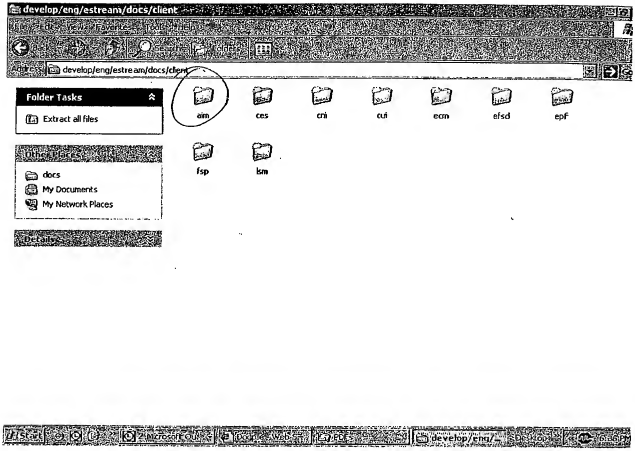
  Cross-component testing for the Builder program is described in the Package Manager low-level design document.

Upgrading/Supportability/Deployment design

The Profile Manager logs all error messages to a predefined file common to all components of the Builder program. Every Builder component prints the error message along with its component name. This allows the user of the Builder program to quickly track down any problem during the Building of a new eStream Set.

Open Issues

- How to automate profiling so the application doesn't require any user intervention? Look into using Rational TestSuite programs.
- Can a subset of Winstone be used for profiling? How do we determine which part of the profile data is more useful to the end-user?
- Should Profile Manager actually create data structures like PrefetchBlocks that require FileNumber assignments? Or should Profile Manager just create the output data without knowning FileNumbers? Then Package Manager associates the file numbers assigned by the Install Monitor with the profile data gathered by the Profiler.


develop/eng/estream/docs/client
File
Edit
View
Favorites
Tools
Help
Back
Search
Folders
Address
develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
Microsoft Out...
Google Web...
PDF
develop/eng/...
Desktop
6:36 PM

[/STANDARD.01]

eStream Application Install Manager Low Level Design

Nicholas Ryan

Version 1.5

[redacted]

Functionality

The Application Install Manager (AIM) is a component of the eStream client executable. It is responsible for installing and uninstalling eStream applications at the request of the License Subscription Manager (LSM). AIM uses the information contained in an AppInstallBlock to prepare the user's system for execution of a given eStream application. It creates registry entries, copies files, and updates the file spoofing database. The user can then launch his application via a local shortcut or a shortcut on the eStream drive. Uninstallation involves undoing all changes made to the user's system by AIM during installation.

Data type definitions

This component uses the AppInstallBlock, but doesn't define it. This is defined in a low-level design document for the Builder component.

The AppInstallBlock is a binary data file with a versioned interface, basically consisting of:

- a header
- a list of files to install or send to the file spoofer
- a list of registry entries to install or remove
- a set of prefetch requests to communicate to the profile/prefetch component
- a set of initial profile data to communicate to the profile/prefetch component (post-version 1.0)
- a comment section
- an embedded DLL that can be loaded and executed for custom install needs
- a section containing a license agreement to be shown to the user

Many of the AIMsc functions take an AIBFileRef as an argument, which is an opaque pointer to a Builder-provided class called 'AppInstallBlock'. This utility class will be shared by the Builder and the AIM code. It encapsulates the complexity of reading from and writing to the various sections of an AppInstallBlock file.

Also, each application has a prefetch data file created for it an install time that is initialized with prefetch data from the AppInstallBlock. This data file is named and located as described in the Component Design section, and consists of a list of critical blocks followed by a list of common blocks. The file begins with the following structure:

```
typedef struct
{
   OTUInt32  numCriticalBlocks;
   OTUInt32  numCommonBlocks;

} PrefetchFileHeader, *pPrefetchFileHeader;
```

The remained of the file consists of a non-padded list of the following structures:

```
typedef struct
{
   OTUInt32  fileNumber;
   OTUInt32  blockNumber;

} PrefetchItem, *pPrefetchItem;
```

The following data types are used in the AIM and AIMsc interfaces:

```
typedef void *AIBFileRef;
```

Error codes are defined in the OTError enumeration, which lives in a file called ot_errors.h.

Interface definitions

Application installation/uninstallation

There are only two functions exposed by AIM, one for application installation, and another for application uninstallation. Only the License Subscription Manager will be calling these functions.

OTError AIMInstallApplication(OTUInt8 appId[16], const char *pathToAIB)

Parameters

appId

[in] The application ID of the eStream application to install.

pathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to install.

Return Values

OT_SUCCESS if all the actions specified in the AppInstallBlock were performed successfully, an error code otherwise.

Comments

None.

OTError AIMUninstallApplication(OTUInt8 appId[16])

Parameters

appId

[in] The application ID of an existing eStream application to uninstall.

Return Values

If the specified application ID is not recognized, or the original AppInstallBlock is not found, an error code will be returned. Otherwise, AIM will make an attempt to undo all of the actions it performed while installing this application. It will return OT_SUCCESS if it undid enough of these actions so that any future installation of the same application will succeed.

Comments

None.

AIM Sub-Component Interface

Much of the functionality required by the AIM design will be useful to the Builder testing framework as well. This functionality will be treated as a sub-component within the AIM component, called AIMsc, and will export a well-defined interface. That interface is defined as follows.

OTError AIMscOpenAppInstallBlock(const char *pathToAIB, AIBFileRef *pAIBFile)

Parameters

pathToAIB

[in] Pointer to a null-terminated string that specifies a path to an AppInstallBlock file to open.

pAIBFile

[out] Returns a reference to an open AppInstallBlock file.

Return Values

OT_SUCCESS if the AppInstallBlock was opened successfully and validated, an error code otherwise.

Comments

The reference returned by this function can be used as a parameter to any of the other functions that take an AIBFileRef.

OTError AIMscCloseAppInstallBlock(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

Return Values

OT_SUCCESS if the close succeeded, an error code otherwise.

Comments

None.

void AIMscGetAIBAppId(AIBFileRef aibFile, OTUInt8 pAIBAppId[16])

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersion

[out] Returns the value of the AppId field in the AppInstallBlock.

Return Values

OT_SUCCESS if the value was successfully retrieved, an error code otherwise.

Comments

None.

void
AIMscGetAIBVersionNo(AIBFileRef aibFile, OTUInt32 *pAIBVersionNo)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBVersionNo

[out] Returns the value of the VersionNo field in the AppInstallBlock.

Return Values

OT_SUCCESS if the value was successfully retrieved, an error code otherwise.

Comments

None.

```
void
AIMscGetAIBShouldReboot(
  AIBFileRef aibFile,
  OTBool     *pAIBShouldReboot)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBShouldReboot

[out] Returns the value of the ShouldReboot flag in the AppInstallBlock.

Return Values

SUCCESS if the value was successfully retrieved, an error code otherwise.

Comments

None.

OTError
AIMscGetAIBAppName(

```
AIBFileRef      aibFile,
char            *pAIBAppName,
OTUInt16        *pSizeAIBAppName)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

pAIBAppName

[out] The value of the ApplicationName field in the AppInstallBlock is copied into the memory pointed to by this address (it will be null terminated).

pSizeAIBAppName

[in, out] On input, should point to the size of the memory at *pAIBApp-Name*. On output, will point to the total bytes needed to hold the entire string if OTERROR_BUFFER_TOO_SMALL is returned, otherwise is undefined.

Return Values

OT_SUCCESS if the value was successfully retrieved, OTER-ROR_BUFFER_TOO_SMALL if the buffer is too small to hold the entire string, or another error code otherwise.

Comments

None.

OTError
AIMscCheckAIBCompatibleOS(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

Return Values

OT_SUCCESS if the OS version was successfully retrieved and if the currently running OS is compatible.

Comments

This function will check if the currently installed operating system and service is compatible with the specified AppInstallBlock (using the compatibility information contained in the AppInstallBlock). If not, it will return OTERROR_INCOMPATIBLE_OS.

```
OTError
AIMscInstallAppFiles(
  AIBFileRef      aibFile,
  HKEY            spoofKey,
  const char      *installLogFile,
  OTBool          *pIsRebootNeeded)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

spoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file copying, FALSE otherwise.

Return Values

OT_SUCCESS if all file install operations succeeded, an error code otherwise.

Comments

This function will perform the file copies and add the file spoofing entries specified in the File section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

This function will not undo file copies or spoof entry additions if it fails. **AIMscUninstallAppFiles** should be called to do so.

```
OTError
AIMscUninstallAppFiles(
   AIBFileRef      aibFile,
   HKEY            spoofKey,
    const char     *installLogFile,
   OTBool          *pIsRebootNeeded)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

spoofKey

[in] An open handle to the registry key where file-spoofing data is stored.

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

pIsRebootNeeded

[out] Returns TRUE if a reboot is needed to complete the file deletions, FALSE otherwise.

Return Values

OT_SUCCESS if enough of the file install operations were reversed so that re-installation will succeed and so that the system is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the file additions and remove the file spoof database entries specified in the install log file.

```
OTError
AIMscInstallAppVariables(
   AIBFileRef      aibFile,
   const char      *installLogFile,
   const char      *varRefCountFile)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

varRefCountFile

[in] A null-terminated string representing the path to the variable refcounting file for this user.

Return Values

OT_SUCCESS if all variable modifications succeeded, an error code otherwise.

Comments

This function will perform the add/remove variable (i.e. registry entry) changes specified in the Variable section of the AppInstallBlock. Changes will be appended to the install log file (it is created if it doesn't already exist).

This function will not undo registry modifications if it fails. **AIMscUninstallAppVariables** should be called to do so.

```
OTError
AIMscUninstallAppVariables(
   AIBFileRef      aibFile,
   const char      *installLogFile,
   const char      *varRefCountFile)
```

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock.

installLogFile

[in] A null-terminated string representing the path to a text file to which change entries should be added.

varRefCountFile

[in] A null-terminated string representing the path to the variable refcounting file for this user.

Return Values

OT_SUCCESS if enough of the variable changes were reversed so that re-installation will succeed and so that the registry is in a consistent state. Otherwise, an error code is returned.

Comments

This function will reverse the add/remove variable (i.e. registry entry) changes specified in the install log file.

*OTError AIMscInstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)*

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

prefetchFile

[in] A null-terminated string representing the path to the prefetch file to be created.

Return Values

OT_SUCCESS if prefetch block installation succeeded, an error code otherwise.

Comments

This function will install the prefetch information contained in the Prefetch section of the AppInstallBlock into *prefetchFile.*

*OTError AIMscUninstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)*

Parameters

AIBFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

pPrefetchFile

[in] A null-terminated string representing the path to the prefetch file to be uninstalled.

Return Values

OT_SUCCESS if prefetch block uninstallation succeeded, an error code otherwise.

Comments

This function will remove the prefetch information stored at *prefetchFile.*

OTError AIMscCallCustomInstall(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Install()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Install()*.

OTError AIMscCallCustomUninstall(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

An error code if the custom code .dll could not be extracted, loaded and called. Otherwise, returns the value returned by the *Uninstall()* function in the custom code .dll.

Comments

This function will extract and load the custom code .dll included in the Code section of the AppInstallBlock, and then call the exported .dll function named *Uninstall()*.

OTError AIMscEnforceLicenseAgreement(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

OT_SUCCESS if the license agreement was successfully displayed and agreed to by the user, OTERROR_USER_CANCELLED, if it was displayed and not agreed to by the user, or an error code if it could not be displayed.

Comments

This function will extract the license agreement text included in the LicenseAgreement section of the AppInstallBlock and display it to the user. The user will be given the option to agree or not agree to the license (probably via a pair of buttons in a dialog). OTERROR_USER_CANCELLED is returned if the users decides not to accept the license agreement.

OTError AIMscDisplayComment(AIBFileRef aibFile)

Parameters

aibFile

[in] An opaque reference to an open AppInstallBlock previously returned by AIMscOpenAppInstallBlock

Return Values

OT_SUCCESS if the comment was successfully displayed, an error code otherwise.

Comments

This function will display to the user the comment included in the Comment section of the AppInstallBlock.

Component design

AIMsc does not have hard-coded knowledge regarding any of the standard registry and file locations used by AIM, which is why the functions in its interface take as inputs specifiers for filenames and base registry locations. Conversely, AIM itself has no knowledge of the internal structure of the AppInstallBlock file, which is why it must call AIMsc functions to work with such files.

Expansion is perform on registry entries and file paths containing certain variables, when they are read from the AppInstallBlock. These variables are defined in the Builder-LLD and will be recognized and expanded by AIM. (This includes file-spoof entries.)

AIM stores its data in the expected places for an eStream client component. All of the data AIM stores is user-specific, so it makes no use of the global locations defined for eStreams.

Registry keys

AIM stores its registry keys and values under:

HKEY_CURRENT_USER\SOFTWARE\Omnishift\eStream\AIM

This key will have its permissions modified so that ordinary users cannot modify the key (but the eStream client service will be given privileges so that it can do so). Here are the subkeys AIM places under this key:

"SpoofEntries"

Spoof entries are placed here. All spoofing is done globally, so there is no need to place it under an eStream-app specific key. Each value under this key is a pair of pathnames as follows:

<old-pathname> (REG_MULTI_SZ)

- <array of spoofed-pathnames>

The array contains one pathname for each installed eStream app that is spoofing the file. The file spoofer currently only spoofs to one of these files at a time, but this corresponds to a kind of refcounting of the spoof entry.

"<AppId>"

Every installed eStream app has its own subkey whose name is a string representation of its AppId, like so: "{00000000-0000-0000-0000-00000000000}". The values stored under each such key are:

AppId (REG_BINARY)
- AppId in binary form (16 bytes)

AppName (REG_SZ)
- name of the application (same as in the AppInstallBlock)

AppInstallBlockPath (REG_SZ)
- path to the AppInstallBlock for the application

AppInstallState (REG_DWORD)
- a value of 0 means app is installed, 1 means install is in progress, 2 mean uninstall is in progress.

Files

AIM stores per-user files at the following path:

(Path to the user's home directory)\Application Data\Omnishift\eStream\AIM

Here is what is stored in this directory:

RegistryRefCounts.dat	- registry refcounts for eStream apps

<AppId GUID string>	- a folder for each installed app

RegistryRefCounts.dat is a file that associates a refcount with every registry key that is added and every value that is modified during an eStream app install. Each null-terminated line in the file is of the form:

<hash of registry key\value path><tab><refcount of registry key\value>

For each installed application, a separate data folder is created. The name of the folder is the AppId of the application in GUID ASCII format, like so: "{00000000-0000-0000-0000-00000000000}". The files stored under each such folder are:

<GUID string>-AIB.dat	- the AppInstallBlock file for the application
<GUID string>-Prefetch.dat	- the prefetch data file for the application
InstallLog.txt	- a generated log of what to do during uninstall

The Prefetch data file is simply an array of PrefetchItem structures (as described in the Data Structures section).

The InstallLog.txt is a list of undoable actions taken during installation. This log will be used during uninstall to determine which files and entries are safe to remove. Each line in the file contains one change, and is of the form:

ADDED or OVERWROTE or SPOOFED FILE <filename> (fully qualified)
ADDED or OVERWROTE KEY <keyname> (fully qualified)
ADDED or OVERWROTE VALUE <valuename> (fully qualified)

AIMInstallApplication Prototype

Installing an eStream application consists of the following steps:

1. Preparing for the installation
2. Displaying a license agreement to the user and having him agree to it
3. Installing all required local files and spoof entries for this app
4. Setting/removing registry entries as required
5. Initializing the prefetch data for this app
6. Performing any required custom installation tasks
7. Displaying the comment to the user if required
8. Completing the installation
9. Rebooting the computer if necessary

AIM's policy is that if it encounters any fatal error during the execution of AIMInstallApplication, it will attempt to undo everything it did before returning. AIM also gracefully handles aborted installs and uninstalls.

Step 1 – Preparing for the installation

First, AIM checks if the application is already installed by looking for an AppId registry key for the specified AppId. If found, then the AppInstallInProgress registry value is checked. If it exists and is 1, the user is asked if he wants to re-install, otherwise, he is asked to restart an aborted or damaged installation. If the user says no, **AIMInstallApplication** cleans up and exits with OTERROR_USER_CANCELLED.

Next, a free disk space check is performed to ensure that enough disk space is available for the install. The available free space must be at least twice the size of the AppInstallBlock to proceed. If not, **AIMInstallApplication** exits with OTERROR_VOLUME_FULL.

Next, an AppId folder is created for the app (described earlier), and the AppInstallBlock file is copied to this folder. Then AppId registry key is created and the four defined values created and initialized. The AppInstallState value in particular is set to 1 to indicate an install is in progress. Then, AIM then opens the AppInstallBlock using **AIMscOpenAppInstallBlock**, and calls **AIMscCheckAIBCompatibleOS** to check for OS compatibility. If any of these operations fail, **AIMInstallApplication** cleans up and exits with an error.

Step 2 – Displaying the license

AIMscEnforceLicenseAgreement is called to display the license text to the user and ask for his agreement. If the functions fails or if the user rejects the agreement, **AIMInstallApplication** cleans up and exits with OTERROR_USER_CANCELLED.

Step 3 – Installing local files

The install log file to be used for this application is created or open and truncated. **AIMscInstallAppFiles** is called to copy the install files to the computer and to create the spoof entries specified in the AppInstallBlock If the function fails, **AIMInstallApplication** cleans up and exits with an error.

If it succeeds, a boolean is returned indicating whether a reboot needs to occur due to shared files being overwritten. This value is remembered for use in step 9.

Step 4 – Modifying the registry

AIMscInstallAppVariables is called to perform the registry modifications specified in the AppInstallBlock. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 5 – Initializing prefetch data

AIMscInstallAppPrefetchFile is called to create and initialize the prefetch file for this application. If the function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 6 – Performing custom install tasks

AIMscCallCustomInstall is called to extract the custom code .dll contained in the AppInstallBlock and to call the *Install()* function it exports. If **AIMscCallCustomInstall** fails, **AIMInstallApplication** cleans up and exits with an error.

Step 7 – Displaying a comment

AIMscDisplayComment is called to display any comment to the user contained in the appropriate section of the AppInstallblock. If this function fails, **AIMInstallApplication** cleans up and exits with an error.

Step 8 – Completing the installation

The AppInstallInProgress registry value is set to 0 to indicate the install is complete. **AIMscOpenAppInstallBlock** is called to close the AIBFileRef opened in step 1, and any handles to open registry keys are also closed.

Step 9 – Rebooting the computer (if necessary)

If **AIMscInstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, no reboot is performed and the application is ready to be run. **AIMInstallApplication** exits returning OT_SUCCESS.

AIMUninstallApplication Prototype

Uninstalling an eStream application consists of the following steps:

1. Preparing for the uninstallation
2. Undoing all additions made to the registry during install
3. Removing all files copied during install and removing spoof entries for this app
4. Deleting the prefetch data for this application
5. Performing any required custom uninstallation tasks
6. Completing the uninstallation
7. Rebooting the computer if necessary

If the uninstallation fails for any reason, the called should tell the user that the uninstall has failed and that he should attempt to re-install the application before trying to uninstall again.

Step 1 – Preparing for the uninstallation

First, AIM checks if the application is already installed by looking for the AppId registry key corresponding to the specified AppId. If not found, then **AIMUninstallApplication** exits with OTERROR_ITEM_NOT_FOUND.

Then, the AppInstallState value is set to 2 to indicate an uninstall is in progress. **AIM-scOpenAppInstallBlock** is called to open the AppInstallBlock at the path specified by the AppInstallBlockPath key. If this fails, then **AIMUninstallApplication** exits with an error.

Step 2 – Undoing registry modifications

AIMscUninstallAppVariables is called to reverse the registry additions made during installation. If the function fails, uninstall cannot proceed safely and **AIMUninstallAp-plication** exits with an error.

Step 3 – Undoing file copies and removing spoof entries

AIMscUninstallAppFiles is called to delete the files copied during install and to remove the spoof entries written then. If the function fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

If it succeeds, a boolean is returned indicating whether a reboot needs to occur due to shared files being overwritten. This value is remembered for use in step 7.

Step 4 – Deleting profile/prefetch data

AIMscUninstallAppPrefetchFile will be called to remove the prefetch data stored for this application. Any failure is ignored.

Step 5 – Performing custom uninstall tasks

AIMscCallCustomUninstall is called to extract the custom code .dll contained in the AppInstallBlock and call the *Uninstall()* function it exports. If **AIMscCallCustomUnin-stall** fails, uninstall cannot proceed safely and **AIMUninstallApplication** exits with an error.

Step 6 – Completing the uninstallation

AIMscGetAIBShouldReboot is called and the return value saved. Then **AIMscO-penAppInstallBlock** is called to close the AIBFileRef opened in step 1, and the AppId folder and all its contents are deleted. The AppId registry key and all its subkeys are de-leted also. Any handles to open registry keys are closed. Any failures here are ignored.

Step 7 – Rebooting the computer (if necessary)

If **AIMscUninstallAppFiles** in step 3 returned a value indicating a user reboot is necessary, or if **AIMscGetAIBShouldReboot** is called and returns a value of TRUE, the user is asked to reboot. Otherwise, the uninstallation is complete. **AIMUninstallApplication** exits returning SUCCESS.

AIMsc Function Prototypes

Prototypes for the AIMsc functions declared earlier are given in this section.

OTError
AIMscOpenAppInstallBlock(const char *pathToAIB, AIBFileRef *pAIBFile)

A class AppInstallBlock object is created, and the appropriate member function is called to open and verify the AppInstallBlock. An opaque pointer to this object is returned in the *pAIBFile* parameter.

OTError
AIMscCloseAppInstallBlock(AIBFileRef aibFile)

The AppInstallBlock class object pointed to by *aibFile* is deleted.

void
AIMscGetAIBAppId(AIBFileRef aibFile, OTUInt8 pAIBAppId[16])

void
AIMscGetAIBVersionNo(AIBFileRef aibFile, OTUInt32 *pAIBVersionNo)

```
void
AIMscGetAIBShouldReboot(
   AIBFileRef       aibFile,
   OTBool           *pAIBShouldReboot)
```

```
OTError
AIMscGetAIBAppName(
   AIBFileRef       aibFile,
   const char       *pAIBAppName,
   OTUInt16         *pSizeAIBAppName)
```

These four functions are trivial. They directly map to method calls on the AppInstall-Block class object pointed to by *aibFile*, which retrieves the associated header field of the AppInstallBlock. (See the interface declaration for **AIMscGetAIBAppName** for details on its calling logic.)

OTError
AIMscCheckAIBCompatibleOS(AIBFileRef aibFile)

This function will call an API such as GetVersionEx (for Windows) to determine the currently running operating system. The OS version is then compared to the OS version stored in the AppInstallBlock by calling a method on the AppInstallBlock object pointed to by *aibFile.*

OTError
AIMscInstallAppFiles(

```
   AIBFileRef        aibFile,
   HKEY              spoofKey,
   const char        *installLogFile,
   OTBool            *pIsRebootNeeded)
```

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the retrieve the entries in the File section of the AppInstallBlock.

The File section is organized as a series of trees, with directories as non-leaf nodes and plain files as leaf nodes. All nodes are stored contiguously according to the pre-order traversal of the trees. The AIMsc code does not parse the section itself, but relies upon the code in the provided AppInstallBlock class to do the unpacking for it.

For every file copied or spoof entry added by the algorithm, an entry is made to the file at *InstallLogFile.*

The algorithm is as follows:

```
while there are file entries to read in the File section
    read an entry

    if entry filename contains Builder/AIM defined variables
        replace variables with local expansions

    if the file node is a spoof entry
        call HandleSpoofEntry()
    else
        call HandleCopyEntry()
```

Here is HandleSpoofEntry():

```
if filename already exists
    if existing version is earlier or version can't be read
```

```
        mark for spoofing
    else // filename does not exist
        create zero-length file at filename
        mark for spoofing

    if marked for spoofing
        create/modify spoof entry under SpoofKey
```

Here is HandleCopyEntry():

```
if filename already exists
    increment shared file ref count in registry
    if existing version is earlier or version can't be read
        mark for copy
else // filename does not exist
    mark for copy

if marked for copy
    attempt to copy node file to client computer
    if copy fails
        tell system to perform copy at reboot
```

The *pIsRebootNeeded* argument will be set to TRUE if any file copies were scheduled to happen at reboot, FALSE otherwise. Additionally, if any spoof entries were added, then an IOCTL will be sent to the spoof driver asking it to reload the spoof database.

The shared file reference count mentioned in the algorithm above is stored in a standard place in the Windows registry. AIM will create or increment this reference count for every non-spoofed file included in the AppInstallBlock (they can all be potentially shared since they will be placed outside of the eStream app directory). Each such file has an associated REG_DWORD value under the key at:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs

Even though the subkey name is called 'SharedDLLs', this key is used by all well-behaved applications to refcount all shared files. The value's name is the path to the file and the value's data is a integer that is the reference count for this file.

```
OTError
AIMscUninstallAppFiles(
  AIBFileRef          aibFile,
    HKEY              spoofKey,
    const char        *installLogFile,
    OTBool            *pIsRebootNeeded)
```

Currently, the *aibFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppFiles** is simple. It iterates over the change entries contained in the log file, and undoes file copies and spoof entry additions when it is safe to do so. Here is the algorithm:

```
while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED or OVERWROTE <filename>"
        decrease refcount of file
        if refcount is 0
            attempt to delete file
            if deletion fails
                tell system to perform deletion at reboot

    else if the entry is of the form "SPOOFED <filename>"
        decrease refcount of spoof entry for this filename
        if refcount is 0
            delete/modify the spoof entry
            if 0-byte placeholder at filename still exists
                delete it
```

A failure to delete a file or to schedule its deletion will not cause **AIMscUninstallAppFiles** to fail.

```
OTError
AIMscInstallAppVariables(
    AIBFileRef        aibFile,
    const char        *installLogFile,
    const char        *varRefCountFile)
```

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the retrieve the entries in the Variables section of the AppInstallBlock.

The Variable section is organized as a series of trees, similar to how the File section is organized. There are two types of nodes, key nodes and value nodes. Registry keys can contain other keys and registry values, while registry values are just name/data pairs that are stored in keys. The AIMsc code does not parse the section itself, but relies upon the code in the provided AppInstallBlock class to do the unpacking for it.

Each entry can be either an add key/value entry, or a delete key/value entry. For every change made, an entry is made to the file at *installLogFile.* Also, registry key and value additions made by eStream are refcounted in the file at *varRefCountFile,* but deletions are not refcounted. (Only deletions during uninstall are refcounted).

The algorithm is as follows:

```
while there are variable entries to read in the Variable section
    read an entry

    if entry name contains Builder/AIM defined variables
        replace variables with local expansions

    if the variable node is a key entry
        call HandleKeyEntry()
    else
        call HandleValueEntry()
```

Here is HandleKeyEntry():

```
if the key name is not fully qualified
    error
if this is an add key entry
    if key doesn't already exist
        create key
else    // it's a delete key entry
    delete the key
increment key refcount
```

Here is HandleCopyEntry():

```
if this is an add value entry
    add the value and its data
    increment value refcount
else    // it's a delete value entry
```

```
      delete the value
```

```
UINT32
OTError
AIMscUninstallAppVariables(
   AIBFileRef          aibFile,
   const char          *installLogFile,
   const char          *varRefCountFile)
```

Currently, the *aibFile* parameter is not even needed since all of the information needed for file uninstall is contained in the log file at *InstallLogFile*. However, it is included to enforce a design decision, which is that the user should have a valid reference to an AppInstallBlock before performing any install/uninstall related actions on an eStream app.

The algorithm for **AIMscUninstallAppVariables** iterates over the change entries contained in the log file, and undoes registry key and value additions when it is safe to do so. The biggest complication is that it must handle COM entries specially, which requires two passes. Here is the algorithm:

```
Pass 1:

while there are change entries in the log file
    read the next entry

if this is an entry of the form "ADDED/OVERWROTE KEY <CLSID
keyname>"
    check the refcount of the executable that provides this CLSID
    if the refcount is > 0
        mark this CLSID as sacred

Pass 2:

while there are change entries in the log file
    read the next entry

    if the entry is of the form "ADDED/OVERWROTE KEY <keyname>"
        decrement keyname refcount
        if keyname refcount is 0
            if keyname is not a COM key with a sacred CLSID value
                delete it and all its subkeys and values
    else if the entry is of the form "ADDED/OVERWROTE VALUE <value-name>"
        decrement valuename refcount
        if valuename refcount is 0
            if value is not part of a COM key with a sacred CLSID value
                delete it
```

A failure to delete one or more registry entries will not cause **AIMscUninstallAppFiles** to fail.

OTError
AIMscInstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the data in the Prefetch section of the AppInstallBlock. The data are written out into the file at *prefetchFile* as an array of PrefetchItem structures. Any existing file at *PrefetchFile* is overwritten.

Next, the Prefetch component is called to set up an association between the new application and its prefetch file.

OTError
AIMscUninstallAppPrefetchFile(AIBFileRef aibFile, const char *prefetchFile)

The file at *PrefetchFile* is deleted. (No call need be made to the Prefetch component.)

OTError
AIMscCallCustomInstall(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the data in the Code section of the AppInstallBlock. This data is written out as a .dll library. This library is loaded and the *Install()* function export is called (and its return value returned).

OTError
AIMscCallCustomUninstall(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the data in the Code section of the AppInstallBlock. This data is written out as a .dll library. This library is loaded and the *Uninstall()* function export is called (and its return value returned).

OTError
AIMscEnforceLicenseAgreement(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the text in the License Agreement section of the AppInstallBlock. The text is displayed to the user in a dialog box with two buttons, 'Agree' and 'Disagree'.

OTError
AIMscDisplayComment(AIBFileRef aibFile)

The appropriate methods in the AppInstallBlock object pointed to by *aibFile* are called to retrieve the text in the Comment section of the AppInstallBlock. The text is displayed to the user in a dialog box.

Testing design

Unit testing plans

AIM will be tested by a program that generates AppInstallBlocks designed to stress the component. AIM will be asked to install the given AIB and if successful, the resulting state of the system will be compared to the expected state had all the files and variables been installed correctly. An uninstall will then be performed and the system state also checked.

The focus of the testing will obviously be on the File and Variable sections. The other sections such as Code and Comments will be stressed also, but their boundary conditions are much simpler.

AIM's ability to gracefully handle aborted installs and uninstalls will also be tested.

Stress testing plans

The program described above can be deliberately tuned to create AppInstallBlocks of unusual size and organization. For example, AppInstallBlocks with thousands of files and registry entries, or files and entries with unusually long names, etc.

Coverage testing plans

In addition to the stress testing, deliberately malformed AppInstallBlocks will be generated by the test program to hit as much error-handling code as possible. AIM's data files and registry entries can also be deliberately mangled to help achieve this effect.

Cross-component testing plans

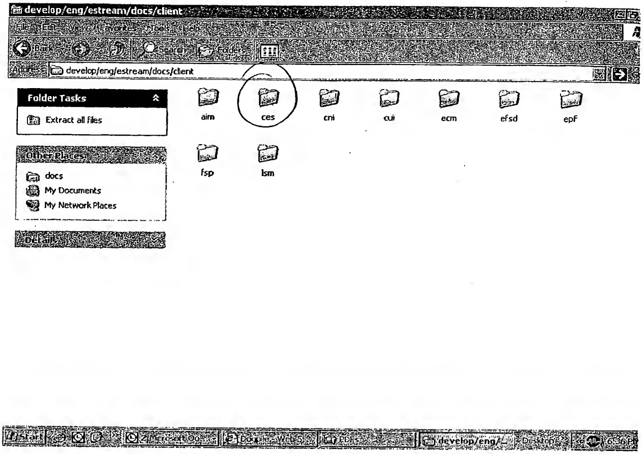
As soon as they are available, Builder-generated AppInstallBlocks will be tested to verify that the AIM is compatible with the Builder's output. As soon as the LSM and a browser plugin are available, the communication path from browser to LSM to AIM will be tested. As soon as the file spoofer is available, compatibility with the file spoof entries that AIM makes will be tested.

Upgrading/Supportability/Deployment design

AIM will make use of the eStream logging facility to record information about errors and other unusual conditions that occur. The log file will be useful for diagnosing problems that occur during testing and in real world situations.

If the AIM component is upgraded, it must still be able to uninstall any eStream applications installed at the time of upgrade. This entails being able to interpret old AIM registry entries and data files, including the AppInstallBlock. This is more a concern for future designers of the AIM component, however.

Open Issues


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back Search Folders
Address develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epF
fsp
lsm
Start
2 Microsoft Of
Google - Web S
PDFs
develop/eng/
Desktop
6:36 PM

eStream Client Installer/Uninstaller & Startup/Shutdown Low Level Design

Anne Holler * [redacted] * *Version 2.9*

Functionality

The Client Installer/Uninstaller [CIN] software is used to setup/remove a client's capability to eStream applications via installing/uninstalling eStream client software. [eStream client software patches are handled via special encapsulated executables distributed with patches & are not discussed herein.] eStream client software consists of:

- kernel-mode drivers [EFSD, FileSpoofer, NoCluster]
- user-mode modules comprising the privileged eStream Client Executable [ECE]:
  - Client EStream Startup/Shutdown [CES]
  - Estream Prefetch Fetch [EPF]
  - License Subscription Manager [LSM]
  - Application Install Manager [AIM]
  - EStream Cache Manager [ECM]
  - Client Network Interface [CNI]
  - Client User Interface [CUI]
- user-mode non-privileged executable eSUser.exe, which is run by an eStream user at log in, at log out, & optionally at other times
- user-mode non-privileged Client Browser Module [CBM] called eSCBM.exe, which fields notification messages from ASP servers [Design reviewers suggested investigating combining eSUser.exe & eSCBM.exe, migrating CUI into the module, & allowing MIME messages to control UI operations]

The Component Design section of the document first addresses CIN.

The Client EStream Startup/Shutdown [CES] software comprises ECE's main; it calls client components' initialization routines, it orchestrates activating various eStream client drivers, and it handles shutting the eStream client software down. The Component Design section of the document next addresses CES.

To understand the proposed operation of the CIN and CES, the reader is expected to be familiar with all client LLD design material.

Data Definitions

CIN is a standalone process and does not share any data definitions with other portions of eStream; registry keys used for communicating data are defined in the Component Design section. CES obtains, verifies, tracks, and communicates information about the eStream user (his windows account name and a handle for his display) for passing to LSM and CUI.

Interface Definitions

Registry keys and IOCTL interfaces are described in the Component Design section. Below are the procedural interfaces.

From CES:

- CNIInitialize(void)
- EPFInitialize(void)
- ECMInitialize(void)
- CUIInitialize(IN HandleToDesktop)
- LSMInitialize(IN WindowsUserName)
- LSMUpdateAllSubscriptionStatus(void)

To CES:

- CESSetMode(IN ModeRequested) -- Modes are STANDBY, READY, ACTIVE

Component Design

Client Installer/Uninstaller [CIN]

This section outlines the operations involved in installation & uninstallation of the four client packages listed above, after describing special attributes of the ECE and CBM that are relevant to CIN. [BTW, to put our installation into perspective, SoftwareWow installs 19 files: 4 EXEs, 1 DLL, 6 SYSs, 2 VXDs, 4 WAVs, and 2 HLP/CNTs.]

Attributes of eStream Client Executable

The ECE is chiefly a performance-critical extension of the eStream client file system and certain parts of ECE have been identified as potentially moving to kernel-mode for performance reasons, including ECM, EPF, LSM and possibly CNI. The ECE contains certain administrative modules for which it is desirable to execute in the context of a separate privileged account, including AIM, LSM, and possibly CES. Several client tasks [eSUser.exe and eSCBM.exe] need to communicate with ECE asynchronously. Given this set of characteristics. ECE will be implemented as a daemon process launched at boot time.

On W2K/WNT, ECE is planned to be a windows service program [ref: MSDN Library \Platform SDK\Windows Base Services\Executables\Services]. ECE starts running when the system boots and it runs under a privileged account to allow it to hide various data for privacy and piracy protection. ECM is not a COM server; it does not contain any interfaces intended to be exported for running as instances inside a COM client. It is also not a COM client; it does not manage compound documents or import functionality from any COM servers. ECE contains two message queues into which client tasks can post messages. One message queue [CBMmessages] handles messages from eSCBM.exe & the other message queue [USRmessages] handles messages from eSUser.exe. [Guidance on selecting between Clipboard, COM, DDE, FileMapping, Mailslot, Pipes, RPC, Sockets, & WM_COPYDATA for IPC implementation is found in MSDN Library \Platform SDK \Windows Base Services \Interprocess Communication\Choosing an IPC Mechanism.]

Attributes of Client Browser Module

The CBM is a standard browser helper application, i.e., it is a Windows32 application that is registered to handle a particular MIME type embedded in HTML. Helper apps are supported in both Internet Explorer 4+ and Netscape Navigator 4+. To register a helper application for handling a certain MIME type, one sets up the following:

- key for mime extension under HKLM\MIME\Database\Content Type
- key under HKLM\SOFTWARE\Classes for class type which application handles
- key under HKCR for file extension which application handles

[BTW, SoftwareWow creates these entries to register a helper app to handle .wow files.]

CBM talks with ECE by placing messages in the CBMmessage message queue.

eStream Client Installation

A standard installation tool [likely InstallShield] will be used to create the installation package. Installation images for distribution on CD-ROM and via the internet will be provided. Uninstallation via the "Add/Remove Programs" will be supported.

BTW, the windows installer is discussed in MSDNLibrary/ Platform SDK/ Management Services/ Setup/Windows Installer. The windows installer is a privileged service. A "managed application" is an application that is installed on a per-machine (not per-user) basis & which requires elevated privileges for installation. The application installation is marked via the ALLUSERS property as to its installation privilege needs. If app is per-machine, but user is unprivileged, installation will fail unless the AlwaysInstallElevated registry key is set.

Installation will be performed under administrative privileges. To handle the situation of residual partial installation due to a failed/aborted previous installation, there is always an [invisible] uninstall-like step that cleans up before an install. eStream client software (and supported eStream applications) expect a client's DLLs to be at a certain revision level; the installation process will check them & if they are not at that level or higher, the installation will include upgrading them. EStream-specific installation activities are described in the paragraphs that follow.

The user is asked where he would like to install eStream-related files, with the default location set to C:\Program Files\Omnishift\eStream. The installation path chosen is written to the registry key HKLM\Software\Omnishift\eStream\InstallPath. A directory for TEMP files is created at InstallPath\temp and the temp path is written to the registry key HKLM\Software\Omnishift\eStream\TempPath. The estream version number is written to HKLM\Software\Omnishift\eStream\Version. The user is asked the drive letter/UNC path at which to mount the eStream file system & this is put into the registry at HKLM\Software\Omnishift\eStream\FileMountLocation.

The kernel-mode drivers (NoCluster, EFSD, FileSpoofer) are installed in the system directory & set up via registry entries for automatic loading at boot time.

- Copy {NoCluster.sys,efsd.sys,fspoof.sys} to %SystemRoot%\system32\drivers
- Set up registry to specify loading at boot time
    - o Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_DRIVER (as appropriate)
        - Set START to SERVICE_SYSTEM_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL

Please note that the names NoCluster.sys & fspoof.sys are used above for the reader's convenience. For the product, we may obscure their purpose via naming them something like OTI001.sys & OTI002.sys.

The ECE is installed as a windows service & set up for automatic loading at boot time.

- Create a special privileged account under which the ECE service will run
- Copy estream.exe to %SystemRoot%\system32\estream.exe
- Set up registry to specify load at boot time, specify to run under privileged accnt
    - o Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_WIN32_OWN_PROCESS
        - Set START to SERVICE_AUTO_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL
        - Set IMAGEPATH to %SystemRoot%\system32\estream.exe

eSUser.exe is copied to location in HKLM\Software\Omnishift\eStream\InstallPath.
eSCBM.exe is copied to location in HKLM\Software\Omnishift\eStream\InstallPath.
eStream user-specific actions are set up:

- During installation, user is asked if he/she would like eStream to be automatically activated at log in to windows (preferred). The answer determines which value (ready|activate) is sent to eSUser.exe when it is run at the user's login.

- User account is set up to run "eSUser.exe ready|activate" whenever user logs in to windows.
- User gets a Start\Program link which runs "eSUser.exe activate". (BTW, both SoftwareWow & NewMoon have Start\Program entries.)
- User account is set up to run "eSUser.exe standby" whenever user logs out of windows.
- Registry keys are set up to hold user's ASP Data Set and Subscription Data Set.

eStream Client Uninstallation

eStream-specific installation activities are undone at uninstall; the files NoCluster.sys, efsd.sys, fspoof.sys, estream.exe, eSUser.exe, & eSCBM.exe are deleted, the relevant registry keys are removed, and the user-specific operations for each eStream user on this client are yanked. We need to decide if apps should be [automatically] uninstalled when user requests that eStream client software be uninstalled.

Client eStream Startup/Shutdown [CES]

The actions of CES need to be understood in the context of the overall operation of the eStream client software; hence, the description of CES below is embedded in a general discussion of the operation of ECE. This is followed by a section considering the issues surrounding usability when eStream client software is in READY (i.e., not yet active) mode for a user.

Operation of ECE

At system boot time, the kernel-mode drivers are loaded, but they are not performing eStream-related activities. ECE is loaded as a service program belonging to the eStream privileged account set up at install time and is ready to receive messages in its message queues. The eStream client software is considered to be in STANDBY mode. No systray icon is displayed.

When an eStream user logs in to the client's console, "eSUser.exe ready|activate" is run and it sends a READY message to ECE on behalf of this user. [Please note that a client system only supports one active eStream user at a time.] When ECE receives the message, it:

- runs initialization routines for each client module that has them [expected order from lowest to highest level: CNIInitialize, LSMInitialize, EPFInitialize, ECMInitialize, CUIInitialize]
- calls the LSM interface that reads in this user's ASP & Subscription data [stored in privileged registry entries] and that performs a synchronize operation
- brings up a systray icon on the eStream user's desktop

The eStream client is now considered to be in READY mode on behalf of this user; it remains in READY mode if eSUser.exe was invoked with the ready parameter.

Whenever eSUser.exe is run with the activate parameter (either automatically at the user's windows login or manually via a Start/Programs entry selection), it sends an ACTIVATE message to ECE on behalf of the user. [A discussion of issues related to eStream being activated automatically in other situations is included below.] The eStream client software moves from READY mode to ACTIVE mode for the user in question, performing the following actions:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to begin eStream-related activities:
    - NoCluster: CES sends IOCTL to activate [or may be active at boot]
    - EFSD: CES sends IOCTL_EFS_START_FS
    - FileSpoofer: CES sends IOCTL_FS_START_SPOOFING
- If any ECE routines need to execute separate initialization on client activation, those routines are called at this point.

When eStream client software is in ACTIVE mode for a user, that user can seamlessly run eStream applications; when it is in READY mode for a user, eStream applications cannot be executed immediately. Issues involving eStream client software automatically transitioning from READY to ACTIVE when a user attempts to run an eStream application or providing a meaningful error message to the user if that automatic transition is not supported are considered in the next section.

ECE may be deactivated automatically at eStream user logout [eSUser.exe standby] or manually on demand via the CUI (mode request for READY). When the ECE is deactivated, the following events are performed:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to stop eStream-related activities:
    - NoCluster: CES sends IOCTL to deactivate [or may continue active]
    - EFSD: CES sends IOCTL_EFS_SHUTDOWN_FS
    - FileSpoofer: CES sends IOCTL_FS_STOP_SPOOFING
- If any ECE routines need to execute separate termination on client deactivation, those routines are called at this point.

ECE brings up a browser to provide web-based ABOUT/HELP information for eStream and to allow the user to interact with his ASP to obtain/modify account-specific data. For eStream 1.0, this communication is handled via the standard mechanism used by currently-available windows application that provides web access, i.e., the user's default browser is invoked in a separate window with a particular URL and set of parameters [decided at 9/7 meeting of Lacky, Bhaven, Manoj, and me]. A common way to accomplish this is to execute a variant of the ShellExecute command.

Automatic Transition from READY to ACTIVE mode Plus Error Handling

If the eStream client software is not active when an eStream user tries to execute an eStream application, it is attractive to either give a meaningful error message or to automatically activate eStream. [To my knowledge, doing either is an eFSD issue; I am including a discussion of the issue here since the topic seems to be open at this time.] To give a meaningful error message, eFSD needs to see the relevant file reference, meaning that either the eStream drive letter needs to exist or eFSD needs to be registered as a UNC provider; otherwise, we may give a somewhat inscrutable error message. To automatically transition from READY to ACTIVE when we see a relevant file reference, we need to ensure that file spoofing is not needed prior to eFSD receiving the reference to the inactive eStream file system for apps of interest. Please note that it is not required that eStream client software be activated automatically if CBM finds it inactive.

Testing Design

Unit and Coverage testing plans

For CIN, an installation/deinstallation of stub versions of the various eStream client software will be developed. For CES, a test harness will be developed to test all CES operational modes (standby, transition to ready, transition to active, transition to inactive, shutdown) and is expected to achieve essentially 100% PFA coverage.

Stress testing plans

CIN stress testing will involve testing a myriad of software and hardware system configurations. CES stress testing will involve testing behavior in the presence of various problems with the eStream client software configuration and for various user situations.

Cross-component testing plans

CIN is expected to be incorporated with the first set of working eStream 1.0 software & installing/uninstalling should form a key step in the daily automated testing of eStream 1.0 going forward. CES is also expected to be included in the first set of working eStream 1.0 software and should also get a daily workout.

Upgrading/Supportability/Deployment Design

Both CIN and CES should be developed with an eye to minimizing issues related to upgrading/supportability/deployment. The bar is very high for these components to contain heavy error checking and reporting.

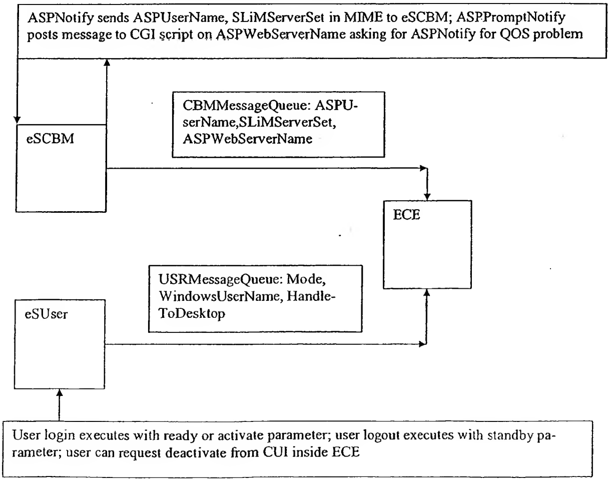
Attachment 1: eStream Client Mode Transitions

Current Mode	Mode Request	Normal Mode Transition
• Standby	Ready	Transition to Ready
• Standby	Active	Transition to Ready, then to Active
• Ready	Active	Transition to Active
• Ready	Standby	Transition to Standby
• Active	Ready	Transition to Ready
• Active	Standby	Transition to Ready, then Standby

Attachment 2: Running scripts at Logon & at Logout

Windows2000 supports executing scripts & programs at user logon & logoff. Registry keys control whether the executions are synchronous or asynchronous wrt logon or logoff & whether the processes interact with the user. To install these processes, one runs start/run/gpedit.msc & chooses UserConfiguration/WindowsSettings/Scripts(Logon/Logoff).

Attachment 3: eStream Client IPC


ASPNotify sends ASPUserName, SLiMServerSet in MIME to eSCBM; ASPPromptNotify posts message to CGI script on ASPWebServerName asking for ASPNotify for QOS problem
eSCBM
CBMMessageQueue: ASPUserName,SLiMServerSet, ASPWebServerName
ECE
USRMessageQueue: Mode, WindowsUserName, HandleToDesktop
eSUser
User login executes with ready or activate parameter; user logout executes with standby parameter; user can request deactivate from CUI inside ECE

Client Installer/Uninstaller & Estream Startup/Shutdown Straw Man

Anne Holler *[redacted]* *Version 1.6*

Introduction

The Client Installer/Uninstaller [CIN] software is used to setup/remove a client's capability to eStream applications via installing/uninstalling eStream client software. [eStream client software patches are handled via special encapsulated executables distributed with patches & are not discussed herein.] eStream client software consists of:

- kernel-mode drivers [EFSD, FileSpoofer, NoCluster]
- user-mode modules comprising the privileged eStream Client Executable [ECE]:
    - Client EStream Startup/Shutdown [CES]
    - Estream Prefetch Fetch [EPF]
    - License Subscription Manager [LSM]
    - Application Install Manager [AIM]
    - EStream Cache Manager [ECM]
    - Client Network Interface [CNI]
    - Client User Interface [CUI]
- user-mode non-privileged executable eSLogin.exe, which is run by an eStream user at log in, at log out, & optionally at other times
- user-mode non-privileged Client Browser Module [CBM] called eSCBM.exe, which fields notification messages from ASP servers

This document first addresses CIN.

The Client EStream Startup/Shutdown [CES] software comprises ECE's main; it calls client components' initialization routines, it orchestrates activating various eStream client drivers, and it handles shutting the eStream client software down. This document next addresses CES.

To understand the proposed operation of the CIN and CES, the reader is expected to be familiar with all client LLD design material.

Client Installer/Uninstaller [CIN]

This section outlines the operations involved in installation & uninstallation of the four client packages listed above, after describing special attributes of the ECE and CBM that are relevant to CIN. [BTW, to put our installation into perspective, SoftwareWow installs 19 files: 4 EXEs, 1 DLL, 6 SYSs, 2 VXDs, 4 WAVs, and 2 HLP/CNTs.]

Attributes of eStream Client Executable

The ECE is chiefly a performance-critical extension of the eStream client file system and certain parts of ECE have been identified as potentially moving to kernel-mode for performance reasons, including ECM, EPF, LSM and possibly CNI. The ECE contains certain administrative modules for which it is desirable to execute in the context of a separate privileged account, including AIM, LSM, and possibly CES. Several client tasks [eSLogin.exe and eSCBM.exe] need to communicate with ECE asynchronously. Given this set of characteristics. ECE will be implemented as a daemon process launched at boot time.

On W2K/WNT, ECE is planned to be a windows service program [ref: MSDN Library \Platform SDK\Windows Base Services\Executables\Services]. ECE starts running when the system boots and it runs under a privileged account to allow it to hide various data for privacy and piracy protection. ECM is not a COM server; it does not contain any interfaces intended to be exported for running as instances inside a COM client. It is also not a COM client; it does not manage compound documents or import functionality from any COM servers. ECE contains two message queues into which client tasks can post messages. One message queue [CBMmessages] handles messages from eSCBM.exe & the other message queue [LGNmessages] handles messages from eSLogin.exe. [Guidance on selecting between Clipboard, COM, DDE, FileMapping, Mailslot, Pipes, RPC, Sockets, & WM_COPYDATA for IPC implementation is found in MSDN Library \Platform SDK \Windows Base Services \Interprocess Communication\Choosing an IPC Mechanism.]

Attributes of Client Browser Module

The CBM is a standard browser helper application, i.e., it is a Windows32 application that is registered to handle a particular MIME type embedded in HTML. Helper apps are supported in both Internet Explorer 4+ and Netscape Navigator 4+. To register a helper application for handling a certain MIME type, one sets up the following:

- key for mime extension under HKLM\MIME\Database\Content Type
- key under HKLM\SOFTWARE\Classes for class type which application handles
- key under HKCR for file extension which application handles

[BTW, SoftwareWow creates these entries to register a helper app to handle .wow files.] CBM talks with ECE by placing messages in the CBMmessage message queue.

eStream Client Installation

A standard installation tool [likely InstallShield] will be used to create the installation package. Installation images for distribution on CD-ROM and via the internet will be provided. Uninstallation via the "Add/Remove Programs" will be supported.

Installation will be performed under administrative privileges. To handle the situation of residual partial installation due to a failed/aborted previous installation, there is always an [invisible] uninstall-like step that cleans up before an install. eStream client software (and supported eStream applications) expect a client's DLLs to be at a certain revision level; the installation process will check them & if they are not at that level or higher, the installation will include upgrading them. eStream-specific installation activities are described in the paragraphs that follow.

The user is asked where he would like to install eStream-related files, with the default location set to C:\Program Files\Omnishift\eStream1.0. The installation path chosen is written to the registry key HKLM\Software\Omnishift\eStream1.0\InstallPath. A directory for TEMP files is created at InstallPath\temp and the temp path is written to the registry key HKLM\Software\Omnishift\eStream1.0\TempPath. The estream version number is written to HKLM\Software\Omnishift\eStream1.0\eStreamVersion.

The kernel-mode drivers (NoCluster, EFSD, FileSpoofer) are installed in the system directory & set up via registry entries for automatic loading at boot time.

- Copy {NoCluster.sys,efsd.sys,fspoof.sys} to %SystemRoot%\system32\drivers
- Set up registry to specify loading at boot time
    - Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_DRIVER (as appropriate)
        - Set START to SERVICE_SYSTEM_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL

Please note that the names NoCluster.sys and fspoof.sys are used above as a convenience to the reader. For the product, we may obscure their functionality via naming them something like OTI001.sys and OTI002.sys.

The ECE is installed as a windows service & set up for automatic loading at boot time.

- Create a special privileged account under which the ECE service will run
- Copy estream.exe to %SystemRoot%\system32\estream.exe
- Set up registry to specify load at boot time, specify to run under privileged accnt
    - Create key under HKLM\System\CurrentControlSet\Services w/ values:
        - Set TYPE to SERVICE_WIN32_OWN_PROCESS
        - Set START to SERVICE_AUTO_START
        - Set ERRORCONTROL to SERVICE_ERROR_NORMAL
        - Set IMAGEPATH to %SystemRoot%\system32\estream.exe

eSLogin.exe is copied to location in HKLM\Software\Omnishift\eStream1.0\InstallPath.

eSCBM.exe is copied to location in HKLM\Software\Omnishift\eStream1.0\InstallPath.

eStream user-specific actions are set up:

- During installation, user is asked if he/she would like eStream to be automatically activated at log in to windows (preferred). The answer determines which value (ready|activate) is sent to eSLogin.exe when it is run at the user's login.
- User account is set up to run "eSLogin.exe ready|activate" whenever user logs in to windows.
- User gets a Start\Program link which runs "eSLogin.exe activate". (BTW, both SoftwareWow & NewMoon have Start\Program entries.)
- User account is set up to run "eSLogin.exe deactivate" whenever user logs out of windows.
- Registry keys are set up to hold user's ASP Data Set and Subscription Data Set.

eStream Client Uninstallation

eStream-specific installation activities are undone at uninstall; the files NoCluster.sys, efsd.sys, fspoof.sys, estream.exe, eSLogin.exe, & eSCBM.exe are deleted, the relevant registry keys are removed, and the user-specific operations for each eStream user on this client are yanked.

Client eStream Startup/Shutdown [CES]

The actions of CES need to be understood in the context of the overall operation of the eStream client software; hence, the description of CES below is embedded in a general discussion of the operation of ECE. This is followed by a section considering the issues surrounding usability when eStream client software is in READY (i.e., not yet active) mode for a user.

Operation of ECE

At system boot time, the kernel-mode drivers are loaded, but they are not performing eStream-related activities. ECE is loaded as a service program belonging to the eStream privileged account set up at install time and is ready to receive messages in its message queues. The eStream client software is considered to be in STANDBY mode. No systray icon is displayed.

When an eStream user logs in to the client's console, "eSLogin.exe ready|activate" is run and it sends a READY message to ECE on behalf of this user. [Please note that a client system only supports one active eStream user at a time.] When ECE receives the message, it:

- runs initialization routines for each client module that has them [expected order from lowest to highest level: CNIStartup, LSMStartup, EPFStartup, ECMStartup, CUIStartup]
- calls the LSM interface that reads in this user's ASP & Subscription data [stored in privileged registry entries] and that performs a synchronize operation
- brings up a systray icon on the eStream user's desktop

The eStream client is now considered to be in READY mode on behalf of this user; it remains in READY mode if eSLogin.exe was invoked with the ready parameter.

Whenever eSLogin.exe is run with the activate parameter (either automatically at the user's windows login or manually via a Start/Programs entry selection), it sends an ACTIVATE message to ECE on behalf of the user. [A discussion of issues related to eStream being activated automatically in other situations is included below.] The eStream client software moves from READY mode to ACTIVE mode for the user in question, performing the following actions:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to begin eStream-related activities:
    - NoCluster: CES sends IOCTL to activate [or may be active at boot]

  - EFSD: CES sends IOCTL_EFS_START_FS
  - FileSpoofer: CES sends IOCTL_FS_START_SPOOFING
- If any ECE routines need to execute separate initialization on client activation, those routines are called at this point.

When eStream client software is in ACTIVE mode for a user, that user can seamlessly run eStream applications; when it is in READY mode for a user, eStream applications cannot be executed immediately. Issues involving eStream client software automatically transitioning from READY to ACTIVE when a user attempts to run an eStream application or providing a meaningful error message to the user if that automatic transition is not supported are considered in the next section.

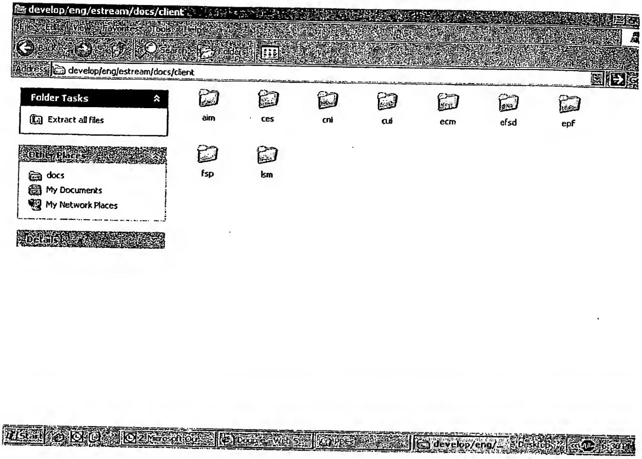
ECE may be deactivated automatically at eStream user logout [eSLogin.exe deactivate] or manually on demand via the CUI. When the ECE is deactivated, the following events are performed:

- CES sends IOCTLs to the kernel-mode drivers, signaling them to stop eStream-related activities:
  - NoCluster: CES sends IOCTL to deactivate [or may continue active]
  - EFSD: CES sends IOCTL_EFS_SHUTDOWN_FS
  - FileSpoofer: CES sends IOCTL_FS_STOP_SPOOFING
- If any ECE routines need to execute separate termination on client deactivation, those routines are called at this point.

ECE brings up a browser to provide web-based ABOUT/HELP information for eStream and to allow the user to interact with his ASP to obtain/modify account-specific data. For eStream 1.0, this communication is handled via the standard mechanism used by currently-available windows application that provides web access, i.e., the user's default browser is invoked in a separate window with a particular URL and set of parameters [decided at 9/7 meeting of Lacky, Bhaven, Manoj, and me]. A common way to accomplish this is to execute a variant of the ShellExecute command.

Automatic Transition from READY to ACTIVE mode Plus Error Handling

If the eStream client software is not active when an eStream user tries to execute an eStream application, it is attractive to either give a meaningful error message or to automatically activate eStream. [To my knowledge, doing either is an eFSD issue; I am including a discussion of the issue here since the topic seems to be open at this time.] To give a meaningful error message, eFSD needs to see the relevant file reference, meaning that either the eStream drive letter needs to exist or eFSD needs to be registered as a UNC provider; otherwise, we may give a somewhat inscrutable error message. To automatically transition from READY to ACTIVE when we see a relevant file reference, we need to ensure that file spoofing is not needed prior to eFSD receiving the reference to the inactive eStream file system for apps of interest. Please note that it is not required that eStream client software be activated automatically if CBM finds it inactive.


develop/eng/estream/docs/client
File Edit View Favorites Tools Help
Back
Search
Folders
Address
develop/eng/estream/docs/client
Folder Tasks
Extract all files
Other Places
docs
My Documents
My Network Places
Details
aim
ces
cni
cui
ecm
efsd
epf
fsp
lsm
Start
Microsoft Out
Dopple Web S
develop/eng/...
Desktop

eStream Client Networking Low Level Design

Dan Arai

Version1.6

[redacted]

Functionality

The Client Network Interface (CNI) provides the interfaces for sending messages to servers and provides threads for receiving responses and dispatching them appropriately. It uses the eStream Messaging Service (EMS) APIs to send and receive various messages to and from the application servers and SLiM servers.

The number of threads in the CNI will depend on the functionality available from the EMS. In particular, more threads are necessary if the EMS provides asynchronous messaging capability (and the CNI uses this interface). The interfaces presented by the CNI are identical for both cases, but the internal organization of the component is not.

The prefetcher will make calls to client networking interfaces (indirectly through ECM-ReservePage) to send requests for pages. Similarly, the LSM will make calls to acquire access tokens and subscription information.

The networking component is responsible for examining the stream of requests to it and deciding when to coalesce multiple page requests into a single request to the server.

The EMS does not provide reliability in the event of server failure. The CNI is responsible for handling server failover and reissuing failed requests on different servers. The CNI abstracts the servers from other parts of the system. Clients of the CNI don't need to specify a particular server to make a request.

Since the client networking component is where timeouts and retries occur, it is the component that controls the policies for how long we wait for a connection to time out and how many times we retry a request before giving up. These parameters will be tunable. Any other parameters of the CNI that make sense to tune will be tunable.

The CNI is also the component responsible for implementing the server selection policy.

Data type definitions

The CNI uses the request structure defined by the ECM.

The CNI maintains an internal queue of messages that must be sent to servers. This queue is not exposed outside of the CNI. Like the ECM request queue, this queue will be maintained as a circular, doubly-linked list.

```
typedef struct _NWRequest
{
     NWRequestType type;
     union {} parameters; /* params, depends on type */
     struct _NWRequest *next;
     struct _NWRequest *prev;
} NWRequest;

typedef enum
{
     CNI_PAGE_READ,
     CNI_ACQUIRE_ACCESS_TOKEN,
     CNI_GET_LATEST_APP_INFO,
     CNI_RENEW_ACCESS_TOKEN,
     CNI_RELEASE_ACCESS_TOKEN,
     CNI_REFRESH_APP_SERVER_SET,
     CNI_GET_SUBSCRIPTION_LIST
} NWRequestType;
```

The CNI provides an enumeration of the parameters that can be tuned. This enumeration is expected to grow as the number of tunable parameters grows.

```
typedef enum
{
     CNI_NUM_RETRIES,
     CNI_TIMEOUT,
     CNI_PROXY_ADDRESS,
     CNI_EFFECTIVE_BANDWIDTH
} NWTunableParameter;
```

Related Components

The prefetcher and LSM call on the CNI to send requests to the app and SLiM servers. The CNI makes calls to the ECM and LSM to inform them of responses that have come back from the server. The CNI will also make calls to EFSD interface functions when pages come back that satisfy EFSD requests.

Interface definitions

CNIGetPage

eStreamStatus CNIGetPage(
   IN ApplicationID *app*,
   IN EStreamPageNumber *page*
);

CNIGetPage is the interface used by the ECM function **ECMReservePage** to request that a page be sent by the server. (**ECMReservePage** is called indirectly by the prefetcher.) Note that no distinction is made between prefetches and demand fetches. To prevent race conditions or deadlock, the requested pages must already be marked as "in flight" in the index, and any requests for these pages from the EFSD must already be on the "in flight" queue before calling this interface.

The CNI is responsible for selecting a server to direct this request to, and resending in the event of network or server failure. It will coalesce requests for multiple pages from the same application into a single request to the server.

CNIGetSubscriptionList

eStreamStatus CNIGetSubscriptionList(
    IN string *Username,*
    IN string *Password*
);

CNIGetSubscriptionList enqueues a request to acquire a subscription list from a SLiM server. When the subscription list is returned by the server, the client response thread will notify the LSM of the returned data via a callback defined in the LSM document.

CNIGetLatestApplicationInfo

eStreamStatus CNIGetLatestApplicationInfo(
    IN uint128 *SubscriptionID*
);

CNIGetLatestApplication enqueues a request to get the latest application information for a particular app. When the server returns the result, the CNI will notify the LSM of the returned data via a callback defined in the LSM document.

CNIAcquireAccessToken

eStreamStatus CNIAcquireAccessToken(
    IN uint128 *SubscriptionID,*
    IN string *Username,*
    IN string *Password*
);

CNIAcquireAccessToken will cause the CNI to contact a SLiM server to retrieve an access token. The CNI is responsible for issuing retries if no response is received for a request. The CNI will call the appropriate LSM callback function when the data come back.

CNIRenewAccessToken

eStreamStatus CNIRenewAccessToken(
    IN AccessToken *Token,*
    IN string *Username,*
    IN string *Password*
);

CNIRenewAccessToken will enqueue a request for access token renewal. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIReleaseAccessToken

eStreamStatus CNIReleaseAccessToken(
    IN AccessToken *Token,*
    IN string *Username,*
    IN string *Password*
);

CNIReleaseAccessToken will enqueue a request for releasing an access token. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

CNIRefreshAppServerSet

eStreamStatus CNIRefreshAppServer(
    IN AccessToken *Token,*
    IN uint32 *BadQOS,*
    IN uint32 *NoService*
);

CNIRefreshAppServerSet will enqueue a request for refreshing the app server set. When the response comes back, the CNI will dispatch the returned data to the appropriate LSM callback routine.

The client networking component will also have routines for getting and setting tunable parameters.

CNISetParameter

eStreamStatus CNISetParameter(
    IN NWTunableParameter *type,*
    IN void **value*
);

CNISetParameter sets a parameter. The actual type of *value* is determined by *type*.

CNIGetParameter

eStreamStatus CNIGetParameter(
    IN NWTunableParameter *type,*
    OUT void **value*
);

CNIGetParameter queries the current value of a parameter. The actual type of *value* is determined by *type*.

Component design

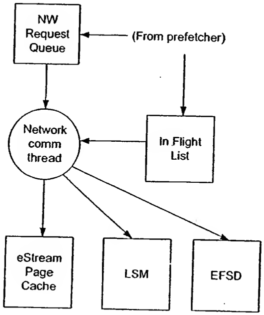
The internal organization of the client networking depends on the mechanisms available from EMS. Internally, the CNI interface functions put requests on a queue, and one or more threads services these requests by using the EMS to send messages to servers.

Synchronous Server Calls

If EMS only provides a synchronous messaging service, a single thread will be used to perform all necessary actions. The CNI interfaces will put appropriate requests on the network request queue. They will also wake up the network communication thread, if necessary.

The network communication thread's job is relatively simple. When it wakes up, it performs the following tasks:
- choose a set of requests to be coalesced and remove these from the request queue
- retrieve a server set via LSMGetAppServerSet or LSMGetSLiMServerSet, and choose a particular server for this request
- make a synchronous EMS call to send the request
- dispatch the response to the appropriate LSM or ECM callback

If the synchronous messaging mechanism becomes a performance bottleneck, we can have multiple network communication threads to increase concurrency.



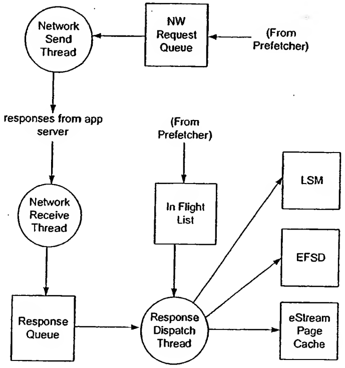
Asynchronous Server Calls

The asynchronous case is a little bit more complex. Because of the proposed asynchronous call architecture, the client NW requires three threads. The CNI interfaces work just as they do in the synchronous case. They put requests on the network request queue, and wake up the network send thread. However, the actions performed by the CNI's worker threads differ in the asynchronous model.

The network send thread is periodically awoken, and it coalesces requests off the NW request queue and sends them to the server. Unlike in the synchronous model, this thread does not synchronously wait for the request to come back from the server. Instead, it simply sends requests until the queue is empty, then goes back to sleep.

The network receive thread waits for responses to come back from any server. Because of the EMS's asynchronous call implementation details, this thread posts returned data to a queue of responses to be handled by another thread. The network receive thread is also responsible for handling timeouts and reissuing those network requests on different servers.

Finally, the response dispatch thread pulls responses off the response queue, and handles the work of dispatching them appropriately.



Handling Network Failure
When the client networking component is notified of a message failure by the EMS, the client worker thread will attempt to reissue the request on a different server.

Coalescing Multiple Requests
The CNI will coalesce multiple page requests that come from the LSM into a single request to an application server. Multiple pages requests for the same application may be coalesced. No other types of requests may be coalesced, including page requests for dif-

ferent applcations. The CNI will not produce requests larger than the maximum allowed by the application server.

Handling Persistent Failures
There will be some persistent failures that will result in the network being unable to fulfill page requests in a timely fashion. This may be due to network or server failure. (These may be indistinguishable from the CNI's point of view.) When the CNI has failed to satisfy a request for a certain amount of time, it will need to ask the user if he wants it to continue retrying, or if it should let the application terminate. It will do this via the CUIAskUserYesNo() interface. The client software control panel should include an option to always wait until the server is available, and never ask the user if he wants the application terminated.

Testing design

Unit testing plans

The testing harness for the networking component will be a set of dummy EMS drivers and a dummy NW client. The dummy EMS driver will be capable of performing a variety of actions, including returning appropriate responses, returning inappropriate responses, and timing out without any response. The dummy NW client will have knowledge about the expected EMS behavior, and will verify that the data it gets back from the network component are as expected.

Stress testing plans

Failure testing plans

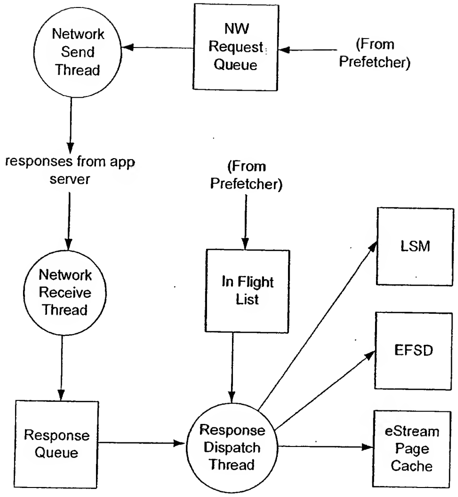
The client NW is the sole component responsible for implementing server failover. In order to test this code, it is necessary to implement a server with predefined bad behavior. The server failure modes that must be tested include
- server that accepts a connection on a socket but doesn't respond to any requests
- server that closes the socket before sending a response
- server that closes the socket in the middle of a response
- server that sends a partial response and then just stops
- server that satisfies n requests then closes the socket or refuses to service more

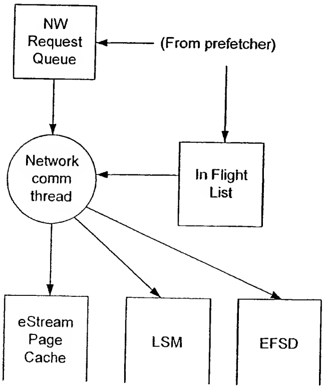
It is important that we cover scenarios that look like network failures and ones that look like server failures. (Are there other failure modes that are interesting?)

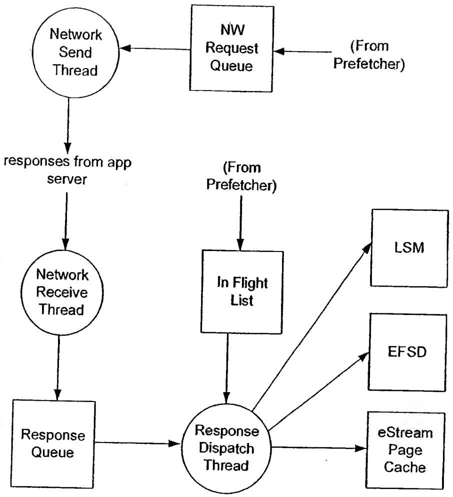
Cross-component testing plans

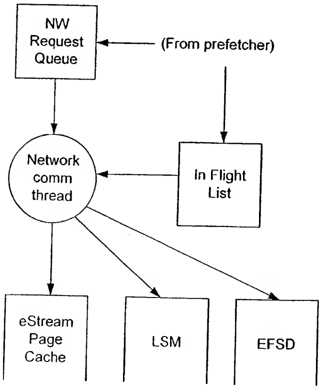
Cross-component testing of the client NW includes integration testing with the EMS, the LSM, and the prefetcher. Testing with the EMS can be performed in a manner similar to unit testing in conjunction with a specially written server. Testing with the LSM or prefetcher can be performed in isolation by writing drivers for either the LSM or prefetcher, and using a dummy or real EMS. I'm not sure if this sort of testing is worth the effort to write the appropriate harnesses. Verifying the output of such a combined system is certainly trickier than testing any component in isolation.

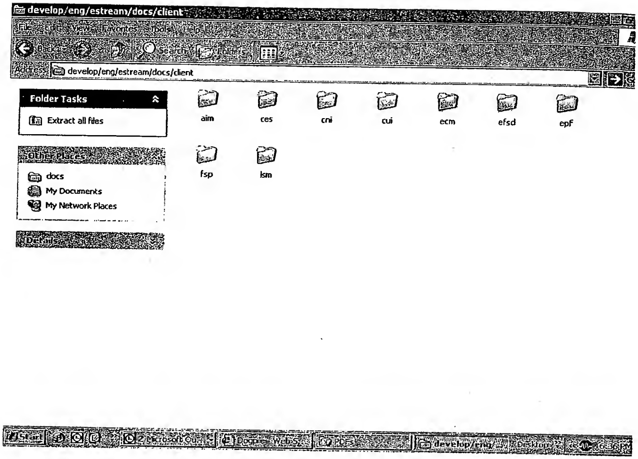
Open Issues


Network Send Thread
NW Request Queue
(From Prefetcher)
responses from app server
(From Prefetcher)
Network Receive Thread
In Flight List
LSM
EFSD
Response Queue
Response Dispatch Thread
eStream Page Cache


NW Request Queue
(From prefetcher)
Network comm thread
In Flight List
eStream Page Cache
LSM
EFSD


Network Send Thread
NW Request Queue
(From Prefetcher)
responses from app server
(From Prefetcher)
Network Receive Thread
In Flight List
LSM
EFSD
Response Queue
Response Dispatch Thread
eStream Page Cache


NW Request Queue
(From prefetcher)
Network comm thread
In Flight List
eStream Page Cache
LSM
EFSD



[/STANDARD.01]

[redacted]

eStream 1.0 Client User Interface Low Level Design

*Anne Holler *[redacted]* Version 2.3*

Functionality

This document represents the low level design of the Client eStream User Interface [CUI], an eStream client module that:

- supports asynchronous communication from various eStream client components to the user, optionally soliciting responses
- provides a visual indication to the user that the eStream client software is running, via the presence of the eStream systray icon
- displays ABOUT and HELP information to the eStream client user, including a link to a web site with FAQs and access to support with respect to eStream
- interfaces with a (planned but not yet designed) client ASP user interface, which allows the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT and HELP information, etc, from the client without requiring that the user manually point his browser at the ASP website and login
- allows an eStream user to request a variety of [optional/advanced] eStream client management functions (described below)

Please note that the eStream CUI is not used to launch eStream applications; eStream applications are launched (like locally-installed applications) via Start/Programs or via desktop icons. The eStream CUI is considered to be a utility interface (in Windows UI parlance) and will follow the Windows UI conventions of this category of program.

Data definitions

eStream client software will be developed in a manner which facilitates porting its user messages and interfaces to languages other than English. Since the CUI is responsible for displaying messages and help text to client users, it is the natural module to "own" the issue of localization.

Windows Resource Files will be used for all user messages; CUI's callers will pass handles to messages they wish displayed. All user displays/menus/bitmaps will be developed in a manner consistent with the guidelines outlined in "Developing International Software for Windows 95 and Windows NT" by Nadine Kano [MSDN Online Library]. Please note that eStream 1.0 is targeted to the English language market, and translation of the Resource Files to other languages is not planned to be completed prior to first product shipment.

The AppName, VersionName, & Message info supplied in the eStreamAppInfo are in the same language as the version of the app (i.e., Japanese Word => Japanese).

Interface definitions

Client Component Acronyms

- CES: Client EStream Startup
- CNI: Client Network Interface
- CUI: Client User Interface
- ECM: EStream Cache Manager
- EPF: EStream PreFetch/Fetch
- LSM: License Subscription Manager

From <various client components> to CUI:

- **CUIInformUser**(IN Message, IN OptionalHelpInfo)
- **CUIAskUserYesNo**(IN Message, IN OptionalHelpInfo, IN CheckAskAgain, OUT Response, OUT DontAskAgain)
- **CUIAskUserPassword**(IN Message, OUT Response, OUT Retain)

From CUI to CES:

- CESSetActivate(IN Boolean ActivateAtLogin)
- CESDeactivate(VOID)

From CUI to LSM:

- LSMGetAspList(OUT NumAsps, OUT AspID[])
- LSMGetAspInfo(IN AspID, OUT ASPWebServerName, OUT UserName)
- LSMUpdateAllSubscriptionStatus(VOID)
- LSMGetAppList(IN AspID, OUT NumApps, OUT AppID[])
- LSMGetAppInfo(IN AppID, OUT AppName, OUT VersionName, OUT Message, OUT AppInstallStatus, OUT AppUpgradeStatus)
- LSMInstall(IN AppID, OUT InstallStatus)
- LSMUninstall(IN AppID, OUT UninstallStatus)
- LSMUpgrade(IN AppID, OUT UpgradeStatus)

From CUI to ECM:

- ECMSetCacheInfo -- causes ECM to recheck registry for cache size
- ECMGetCacheInfo

From CUI to EPF:

- EPFPrefetchSet(IN OnOff)

From CUI to CNI:

- CNIGetParameter(IN ParmName, OUT ParmValue)
- CNISetParameter(IN ParmName, IN ParmValue)

Component design

Asynchronous Messages to eStream User

There are several (hopefully rare) situations that arise asynchronously, about which components of the eStream client software inform the user. In some cases, these situations necessitate user response. CUI delivers asynchronous messages via pop-up dialog boxes. If no user input is needed, other than confirmation that the user has seen the message, the box displays text with a simple Continue button **[CUIInformUser]**. If the message is a question to which a yes/no response from the user is sought, the pop-up dialog box contains Yes and No buttons with the recommended selection emphasized **[CUIAskUserYesNo]**. If the message requires that the user enter a password, the pop-up contains an edit box in which character echoing is shielded **[CUIAskUserPassword]**. Help buttons with supporting amplifying passages are supplied in situations in which extra information can aid user understanding.

eStream Systray Icon

EStream client software can be activated automatically at user login, manually via a Start/Program entry, and automatically when an EFSD request arrives and EFSD is inactive. [Please see the CES design materials for more information on activation.] The eStream client software must be active for the user to execute eStream applications; the presence of the systray icon indicates that the software is active. Right-clicking the icon engenders a pop-up menu from which the user can reach about/help info, ASP client account interfaces, and eStream client management functions.

EStream About/Help Information

The chief external interface involved in eStream-specific ABOUT+HELP is launching a browser and passing arguments to it. The user's default browser is used for this. Please note that the user's default browser must be one of the browsers specified as required for eStream 1.0 (IE4 or higher, Navigator 4 or higher).

ASP Client Account Interfaces

It is beyond the scope of this document to describe the interfaces with a (planned but not yet designed) client ASP user interface, which allows the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT+HELP information, etc., from the client without requiring that the user manually point his browser at the ASP website and login. However, the CUI implementation includes bringing up a browser and passing arguments to it, which is key to supporting the success of this future functionality.

EStream Client Management Functions

The typical user should not normally need to use the eStream client management functions available from the CUI. Subscription information is synchronized automatically, applications are installed/uninstalled when the subscription information is synched, eStream client software is activated at login & deactivated at logout, the cache is right-sized, etc. The functions included in this section are supplied for use in special circumstances and/or in support situations.

eStream client management functions include the following:

- For ASP accounts known to this client:
  - View account information **[LSMGetAspList, LSMGetAspInfo]**
  - Request synchronization of application subscription information [handled automatically at client eStream activation, included here for flexibility] **[LSMUpdateAllSubscriptionStatus]**
- For application subscriptions known to this client:
  - View subscription information [this may engender a synch] **[LSMGetAppList, LSMGetAppInfo]**
  - Install subscribed application [handled automatically at synch time, included here for flexibility] **[LSMInstall]**
  - Uninstall application [handled automatically at synch time, included here for flexibility] **[LSMUninstall]**
  - Upgrade application to latest version [handled automatically, flexibility] **[LSMUpgrade]**
- For eStream client software:
  - Specify automatic activation at login vs manual activation via program/start [it is proposed that we assume the former is always the case] **[CESSetActivate]**
  - Any general user preference wrt terminating app for very slow network response (always wait vs. terminate if too slow)
- For eStream client cache [primarily for users who do not allow eStream to right-size their cache]:
  - Display current size and utilization [obtained from registry, **ECMGetCacheInfo**]
  - Set size: allow eStream to right-size vs. manually increase/decrease size **[ECMSetCacheInfo]**
  - Disable/enable prefetching **[EPFPrefetchSet]**
- For eStream client network interface:
  - Display recent eStream effective bandwidth [possibly displayed as hover-over on eStream systray icon] **[CNIGetParameter]**
  - View/Set proxy IP address [may use info from control panel internet options widget] **[CNIGetParameter, CNISetParameter]**
  - View/Set connection time-out value **[CNIGet/SetParameter]**
  - View/Set number retries value **[CNIGet/SetParameter]**
- Deactivate eStream **[CESDeactivate]**

Testing design

Unit/Coverage testing plans

For the eStream client management functions and the ABOUT+HELP displays, a UI scripting tool like Rational's Visual Test will be used to automate testing. Unit testing will involve creating driver keystrokes that exercise all menu selections and stubs of other client modules that will report as many asynchronous error conditions as possible. Using Rational's PureCov tool, PFA coverage should be as close to 100% as possible.

Cross-component testing plans

With respect to the eStream client management functions, the CUI has many interfaces with LSM, and hooking the two together is a win-win in terms of facilitating the testing of both; this combination will be the first cross-component integration. CUI's unit test scripts are expected to continue to be useful in driving the testing of the CUI/LSM combination. Other client management interfaces will be added and tested as the associated components are completed, with continued emphasis on ensuring that asynchronous error messages are provoked.

Stress testing plans

Need/strategy unclear.

Upgrading/Supportability/Deployment design

A chief motivation for many of the user interfaces in the CUI design, particularly in the eStream client management functions area, is user support and deployment support.

Implementation Issues

- Investigate obtaining ProxyIPAddress from control panel internet options widget. [BTW, why aren't our competitors doing this?]
- Some items accessed by CUI are per-user (ASP, applications, activation) and some items are per-client (cache size, network interface). We need to make sure this is clearly communicated in the UI.
- API needs to be available so that every button/widget can be manipulated programmatically to facilitate testing.

eStream 1.0 Client eStream User Interface Straw Man

Anne Holler *[redacted]* *Version 1.3*

Introduction

This document presents background information related to the Client eStream User Interface [CUI], an eStream client module that:

- supports asynchronous communication from various eStream client components to the user, optionally soliciting responses
- provides a visual indication to the user that the eStream client software is running, via the presence of the eStream systray icon
- displays ABOUT and HELP information to the eStream client user, including a link to a web site with FAQs and access to support with respect to eStream
- interfaces with a (planned but not yet designed) client ASP user interface, which would allow the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT and HELP information, etc, from the client without requiring that the user manually point his browser at the ASP website and login
- allows an eStream user to request a variety of [optional/advanced] eStream client management functions (described below)

The document is intended to provide a baseline for discussions prior to proceeding with a detailed low-level design.

Please note that the eStream CUI is not used to launch eStream applications; eStream applications are launched (like locally-installed applications) via Start/Programs or via desktop icons. The eStream CUI is considered to be a utility interface (in Windows UI parlance) and will follow the Windows UI conventions of this category of program.

Asynchronous Messages to eStream User

There are several (hopefully rare) situations that arise asynchronously, about which components of the eStream client software will inform the user. In some cases, these situations necessitate some user response. CUI delivers asynchronous messages via pop-up dialog boxes. If no user input is needed, other than confirmation that the user has seen the message, the box will display the text along with a simple Continue button [**CUIInformUser**]. If the message is a question to which a yes/no response from the user is sought, the pop-up dialog box will contain Yes and No buttons with the recommended selection emphasized [**CUIAskUserYesNo**]. If the message requires that the user enter a password, the pop-up will contain an edit box in which character echoing is shielded [**CUIAskUserPassword**]. Help buttons with supporting amplifying passages are supplied in situations in which extra information can aid user understanding.

Please note that the eStream client software is expected to be developed in a manner which facilitates porting its user messages and interfaces to languages other than English. Hence, messages are to be obtained from a catalog or resource file to ease translation.

eStream Systray Icon

The eStream client software must be active for the user to execute eStream applications; the presence of the systray icon indicates that it is active. Right-clicking the icon engenders a pop-up menu from which the user can reach help/about info, ASP client account interfaces, and eStream client management functions.

EStream About/Help Information

The chief external interface involved in eStream-specific ABOUT+HELP is launching a browser and passing arguments to it. It is expected that the browser utilized for this feature will be Internet Explorer; if so, we will require that Internet Explorer not be uninstalled from the user's windows platform (which is difficult to do, as demonstrated at Microsoft's antitrust trial). Please note that the user can still employNetscape Navigator to connect to the ASP, as the eStream 1.0 requirements document specifies; this is a separate issue of what browser is used for the client pass-through to ASP account functionality.

ASP Client Account Interfaces

It is beyond the scope of this document to describe the interfaces with a (planned but not yet designed) client ASP user interface, which weill allow the user to view ASP billing information, view/change ASP account information, subscribe/unsubscribe applications, receive ASP-specific ABOUT+HELP information, etc, from the client without requiring that the user manually point his browser at the ASP website and login. However, the CUI implementation includes bringing up a browser and passing arguments to it, which is key to supporting the success of this future functionality.

EStream Client Management Functions

The typical user should not normally need to use the eStream client management functions available from the CUI. Subscription information is synchronized automatically, applications are installed and uninstalled when the subscription information is synched, eStream client software is activated at login, the cache is right-sized, etc. The functions included in this section are supplied for special circumstances and/or support situations.